numbers of simultaneous users; e.g. millions, with access to a large number; e.g. thousands, of applications which may include pre-created, interactive text/graphic sessions; and more particularly, to a computer network in which the interactive text/graphic sessions are comprised of pre-created blocks of data and program instructions which may be distributed downwardly in the network for use at a software enhanced user computer terminal that reduces processing demand on the higher-level network elements, thus permitting the higher-level elements to function primarily as data supply and maintenance resource, and, thereby, reduce network complexity, cost and response time.

Interactive computer networks are not new. Traditionally they have included conventional, hierarchical architectures wherein a central, host computer responds to the information requests of multiple users. An illustrative example would be a time-sharing network in which multiple users at remote terminals log onto a host computer having data and software resource which sequentially receives user data processing requests, executes them and supplies responses back to the users.

B2

While such networks have made the processing power of large computers available to many users, the problem has been that in such networks, the host is required to satisfy all the user data processing requests. This, however, creates processing bottlenecks at the host which tax the host resources and require increased computing power; i.e., bigger and more complex computer facilities, to meet demand as the number of users is increased. Still further, occurrence of such bottlenecking also increases

network response time to unacceptable levels as the number of users is increased.

The size and complexity of the network host, however, is particularly critical in the case of commercial interactive computer network recently introduced to offer large number of the general public text and graphics information to permit not only at home shopping and financial management such as banking and bill paying, but also the providing of information relating to entertainment, business and personal matters.

As can be appreciated, in such state of the art information and shopping networks, the network must be able to provide the information and shopping services with a minimal amount of network resources in order to maintain the capital investment in the network at a level that renders the services economical to use. Unlike military and governmental networks in which capital investment is a secondary concern because of the importance of the service provided, in commercial information and shopping services, the capital investment in the network resources must be kept low in order to make the network affordable both to the users and those who would rely on the network as a channel of distribution for the goods and/or services. Further, in addition, to maintaining capital investment at a minimum, it is also desirable to maintain network response time at a minimum in order to not only capture and hold the user's attention, but also quickly free the network to satisfy the requests of other users.

As noted, this ability to satisfy requests with minimal network resources is required to enable the network to serve large numbers of users and, thereby, render the network economical to use.

While conventional, previously known time-sharing network designs have attempted to alleviate host completely and response time problems by providing some processing at the user site; i.e., "smart terminals", still the storage of the principal data and software resources needed for processing at the host continues to create a burden on network complexity and response time which renders the conventional approach unsuited for the large numbers of users required for a commercially viable computer based information and shopping network.

SUMMARY OF INVENTION

Accordingly, it is an object of this invention to provide method and apparatus which permit a very large number of users to obtain access to a large number of applications which include interactive text/graphic sessions that have been created to enable the users to obtain information and transactional services such as shopping events.

It is a further object of this invention to provide method and apparatus which permit the data and programs necessary to support applications including interactive text/graphic sessions to be distributed over a computer network.

It is a still further object of this invention to provide software that will enable a conventional personal computer to be coupled to a computer network to establish a reception system suitable for supporting applications which include interactive text/graphic sessions created to enable the user to obtain information and conduct shopping events.

It is yet another object of this invention to provide method and apparatus that would permit information and transactional services to be provided to users based upon predetermined parameters such as user demographics and/or locale.

It is yet another object of the invention to provide method and apparatus capable of collecting data regarding usage of the network and to condition the applications and the included text/graphics sessions based upon the reactions to the applications by the users.

Briefly, to achieve the above and other objects and features, the invention includes method and apparatus for providing interactive applications containing text and graphics at the monitor of a personal computer, wherein the personal computer has been configured as a reception system by the inclusion and running of reception system software that enables the reception system so formed to be electronically connected to a network specially adapted to create, maintain and supply databases and portions thereof containing the applications on demand. In

accordance with its method aspects, the invention includes procedures for formulating objects that have been specially structured to include display and control data and program instructions for supporting the applications at the network reception systems, the objects being pre-created, parceled units of information that may be distributed and stored at lower levels in the network, and particularly the reception system, so as to reduce processing demand on the network higher element, and there by permit the higher elements to function primarily as elements for maintaining and supplying the database information.

Further, in preferred form, the method aspect of the invention, features use in combination with the specially formulated objects, of specially structured messages that harmonize and facilitate communications between the different elements of the network and computing elements external to the network that may be called upon to supply information to support the applications.

CANCELED

Also in preferred form, the method aspect of the invention features specially prepared program instructions included within the objects that permit the objects to be executed at the reception system in conjunction with the application software.

Also in preferred form, the invention includes procedures in the form of application software that contain modules that individually and in combination facilitate the execution of objects and the handling of messages at the reception system so

that the interactive sessions may be supported at the reception system.

Still further in its apparatus, aspects the invention includes a reception system comprised of a plurality of types of personal computers combined with the application software for use in the interactive network for displaying information and providing transactional services to a plurality of users, the reception system further comprising in preferred form input means for receiving user inputs; storage means for storing objects containing data or interpretively executable programs, the objects comprising a plurality of partitioned applications; and object processing means, responsive to the input means, for selectively retrieving objects from the storage means and interpreting and executing the partitioned applications.

DESCRIPTION OF THE DRAWINGS

The above and further objects, features and advantages of the invention will become clear from the following more detailed description when read with reference to the accompanying drawings in which:

FIG. 1 is a block diagram of the interactive computer network in accordance with the invention;

FIG. 2 is a schematic diagram of the network illustrated in FIG. 1;

Figures 3(a) and 3(b) are plan views of a display screen presented to a user in accordance with the invention;

Figures 4(a), 4(b), 4(c) and 4(d) are schematic drawings that illustrate the structure of objects, and object segments utilized within the interactive network in accordance with the invention;

FIG. 5(a) is a schematic diagram that illustrates the configuration of the page template object in accordance with the invention;

FIG. 5(b) is a schematic diagram that illustrates page composition in accordance with the invention;

FIG. 6 is a schematic diagram that illustrates the protocol used by the reception system to support user applications in accordance with the invention;

FIG. 7 is a schematic diagram that illustrates major layers of the reception system in accordance with the invention;

FIG. 8 is a block diagram that illustrates native code modules of the reception system in accordance with the invention;

FIG. 9 is a schematic diagram that, illustrates an example of a partitioned application to be processed by the reception system in accordance with the invention;

FIG. 10 illustrates generation of a page with a page processing table in accordance with the invention;

FIG. 11 is a table that lists the TBOL verbs in accordance with the invention; and

FIG. 12 is a block diagram of the reception system in accordance with the invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

GENERAL SYSTEM DESCRIPTION

With reference to Fig. 1, the invention includes a reception system (RS) 400 within interactive computer network 10 for displaying information and providing transactional services. In this arrangement, a plurality of users each access network 10 with a conventional personal computers, for example one of the IBM-compatible or Apple type, which has been provided with applications software in accordance with the invention so as to constitute RS 400.

As shown in Fig. 1, interactive network 10 uses a layered structure that includes an information layer 100, a switch/file server layer 200, and cache/concentrator layer 300 as well as reception layer 401. This structure maintains active application databases and delivers requested parts of the databases on demand

to the plurality of RS 400's, better seen in FIG. 2. As also shown in FIG. 2, cache/concentrator layer 300 includes a plurality of cache/concentrator units 302, each or which serve a plurality of RS 400 units over lines 301. Additionally, switch/file server layer 200 is seen to include a server unit 205 connected to multiple cache/concentrator units 302 over lines 201. Still further, server unit 205 is seen to be connected to information layer 100 and its various elements, which act as means for producing, supplying and maintaining the network databases and other information necessary to support network 10. Continuing, switch/filer layer 200 is also seen to include gateway systems 210 connected to server 205. Gateways 201 couple layer 200 to other sources of information and data; e.g., other computer systems. As will be appreciated by those skilled in the art, layer 200, like layers 401 and 300 could also include multiple servers, gateways and information layers in the event even larger numbers of users were sought to be served.

Continuing with reference to Fig. 2, each RS 400 is seen to include a personal computer 405 having a CPU 410 including a microprocessor (as for example the type made by INTEL Corporation in its X'86 family of microprocessors), companion RAM and ROM memory and other associated elements, monitor 412 with screen 414 and a keyboard 424. Further, personal computer 405 may also include one or two floppy disk drives 426 for receiving diskettes 416 containing application software in accordance with this invention for supporting the interactive sessions with network 10

and diskettes 428 containing operating systems software suitable for the personal computer 405 being used. Personal computer 405 may also include a hard disk drive 420 for storing the application software and operating system software which may be transferred from diskettes 426 and 428 respectfully.

Once so configured, each RS 400 provides: a common interface to other elements of interactive computer network 10; a common environment for application processing; and a common protocol for user - application conversation which is independent of the personal computer type used. RS 400 thus creates a universal terminal for which only one version of all applications on network 10 need be prepared, thereby rendering the applications interpretable by the major brands of personal computers.

RS 400 formulated in this fashion is capable of communication with the host system to receive information containing either of two types of data, namely objects and messages. Objects have a uniform, self-defining format known to RS 400, and include data types, such as interpretable programs and presentation data for display at monitor screen 414 of the user's personal computer. Applications presented at RS 400 are partitioned into objects which represent the minimal units available from the higher levels of interactive network 10 or RS 400. In this arrangement, each application partition typically represents one screen or a partial screen of information, including fields filled with data used in transactions with network 10. Each such

screen, commonly called a page, is represented by its parts and is described in a page template object, discussed below.

Applications, having been partitioned into minimal units, are available from higher elements of network 10 or RS 400, and are retrieved on demand RS 400 for interpretive execution. Thus, not all partitions of a partitioned application need be resident at RS 400 to process a selected partition, thereby raising the storage efficiency of the user's RS 400 and minimizing response time. Each application partition is an independent, self-contained unit and can operate correctly by itself. Each partition may refer to other partitions either statically or dynamically. Static references are built into the partitioned application, while dynamic references are created from the execution of program logic using a set of parameters, such as user demographics or locale. Partitions may be chosen as part of the RS processing in response to user created events, or by selecting a key word of the partitioned application (e.g., "JUMP" or "INDEX," discussed below), which provides random access to all services represented by partitioned applications having key words.

CANCELLED

Objects provide a means of packaging and distributing partitioned applications. As noted, objects make up one or more partitioned applications, and are retrieved on demand by a user's RS 400 for interpretive execution and selective storage. All objects are interpreted by RS 400, thereby enabling applications to be developed independently of the personal computer type used.

Objects may be nested with one another or referenced by an object identifier (object-id) from within their data structure. References to objects permit the size of objects to be minimized. Further, the time required to display a page is minimized when referenced objects are stored locally at RS 400 (which storage is determined by prior usage meeting certain retention criteria), or have been pre-fetched, or in fact, are already used for the current page.

Objects carry application programs and information for display at monitor screen 414 of RS 400. Application program objects, called preprocessor and post-processors, set up the environment for the user's interaction with network 10 and respond to events created when the user inputs information at keyboard 424 of RS 400. Such events typically trigger a program object to be processed, causing one of the following: sending of transactional information to the co-applications in one layer of the network 10; the receiving of information for use in programs or for presentation in application-dependent fields on monitor screen 414; or the requesting of a new objects to be processed by RS 400. Such objects may be part of the same application or a completely new application.

CANCELLED

The RS 400 supports a protocol by which the user and the partitioned applications communicate. All partitioned applications are designed knowing that this protocol will be supported in RS 400. Hence, replication of the protocol in each

partitioned application is avoided, thereby minimizing the size of the partitioned application.

RS 400 includes a means to communicate with network 10 to retrieve objects in response to events occurring at RS 400 and to send and receive messages.

RS 400 includes a means to selectively store objects according to a predetermined storage criterion, thus enabling frequently used objects to be stored locally at the RS, and causing infrequently used objects to forfeit their local storage location. The currency of objects stored locally at the RS 400 is verified before use according to the object's storage control parameters and the storage criterion in use for version checking.

Selective storage tailors the contents of the RS memory to contain objects representing all or significant parts of partitioned applications favored by the user. Because selective storage of objects is local, response time is reduced for those partitioned applications that the user accesses most frequently.

CANCELED

Since much of the application processing formerly done by a host computer in previously known time-sharing networks is now performed at the user's RS, the high elements of network 10, particularly layer 200 has its primary functions routing messages, serving objects, and line concentration. The narrowed functional load of the higher network elements permits many more

users to be serviced within the same bounds of computer power and I/O capability of conventional host-centered architectures.

Network 10 provides information on a wide variety of topics, including, but not limited to news, industry, financial needs, hobbies and cultural interests. Network 10 thus eliminates the need to consult multiple information sources, giving users an efficient and timesaving overview of subjects that interest them.

The transactional features of interactive network 10 saves the user time, money, and frustration by reducing time spent traveling, standing in line, and communicating with sales personnel. The user may, through RS 400, bank, send and receive messages, review advertisements, place orders for merchandise, and perform other transactions.

In the preferred embodiment, network 10 provides information and transaction processing services for a large number of users simultaneously accessing the network via the public switched telephone network (PSTN), broadcast, and/or other media with their RS 400 units. Services available to the user include display of information such as movie reviews, the latest news, airlines reservations, the purchase of items such as retail merchandise and groceries, and quotes and buy/sell orders for stocks and bonds. Network 10 provides an environment in which a user, via RS 400 establishes a session with the network and accesses a large number of services. These services are

specifically constructed applications which as noted are partitioned so they may be distributed without undo transmission time, and may be processed and selectively stored on a user's RS 400 unit.

SYSTEM CONFIGURATION

As shown in FIG.1, in preferred form interactive computer network 10 includes four includes layers: information layer 100, switch and file server layer 200, concentrator layer 300, and reception system 400.

Information layer 100 handles: (1) the production, storage and dissemination of data and (2) the collection and off-line processing of such data from each RS session with the network 10 so as to permit the targeting of information to be presented to users and for traditional business support.

Switch and file server layer 200 and cache concentrator layer 300 together constitute a delivery system 20 which delivers requested data to RS 400 and routes data entered by the user or collected at RS 400 to the proper application in network 10. With reference to FIG.2, the information used in the RS 400 either resides locally at the RS 400, or is available on-demand from the cache concentrator 300 or a the file server 205, via the

gateway 210, which may be coupled to external providers, or is available the from information layer 100.

There are two types of information in the network 10 which are utilized by the RS 400: objects and messages.

Objects include the information requested and utilized by the RS 400 to permit a user to select specific parts of applications, control the flow of information relating to the applications, and to supply information to the network. Objects are self-describing structures organized in accordance with a specific data object architecture, described below. Objects are used to package presentation data and program instructions required to support the partitioned applications of a RS 400, Objects are distributed on demand throughout interactive network 10. Objects may contain: control information; program instruction to set up an application processing environment and to process user or network created events; information about what is to be displayed and how it is to be displayed; references to programs to be interpretively executed; and references to other objects, which may be called based upon certain conditions or the occurrence of certain events at the user's personal computer, resulting in the selection and retrieval of other partitioned applications packaged as objects.

CANCELLED

Messages are information provided by the user or the network and are used in fields defined within the constructs of an

object, and are seen on the user's R.S. monitor, or are used for data processing at RS 400. Additionally, and as more fully described hereafter, messages are the primary means for communication within and without the network. The format of messages is application dependent. If the message is input by the user, it is formatted by the partitioned application currently being processed on RS 400. Likewise, and with reference to FIG.2, if the data are provided from a co-application database residing in delivery system 20, or accessed via gateway 210 or high function system 110 within the information layer 100, the partitioned application currently being processed on RS 400 causes the message data to be displayed in fields on the user's display monitor as defined by the particular partitioned application.

CANCELLED

All active objects reside in file server 205. Inactive objects or objects in preparation reside in producer system 120. Objects recently introduced into delivery system 20 from the producer system 120 will be available from file server 205, but may not be available on cache/concentrator 302 to which the user's RS 400 has dialed. If such objects are requested by the RS 400, the cache controller automatically requests the object from file server 205. The requested object is routed back to the requesting cache/concentrator 302, which automatically routes it to the communications line on which the request was originally made, from which it is received by the RS 400.

The RS 400 is the point of application session control because it has the ability to select and randomly access objects representing all or part of partitioned applications and their data. RS 400 processes objects according to information contained therein and events created by the user on personal computer 405.

Applications on network 10 act in concert with the distributed partitioned applications running on R.S.400. Partitioned applications constructed as groups of objects are distributed on-demand to a user's R.S.400. Partitioned applications represent the minimum amount of information and program logic needed to present a page or window, i.e. portion of a page presented to the user, perform transactions with the interactive network 10, and perform traditional data processing operations, as required, including selecting another partitioned application to be processed upon a user generated completion event for the current partitioned application.

CANCELLED

Objects representing all or part of partitioned applications may be stored in a user's R.S.400 if the objects meet certain criteria, such as being non-volatile, non-critical to network integrity, or if they are critical to ensuring reasonable response time. Such objects are either provided on diskettes 426 together with R.S. system software used during the installation procedure or they are automatically requested by RS 400 when the user makes selections requiring objects not present in RS 400.

In the latter case, RS 400 requests from cache concentrator layer 300 only the objects necessary to execute the desired partitioned application.

Reception system application software 426 is provided for most major brands of personal computers 405, and all partitioned applications are constructed according to a single architecture which each such RS 400 supports. Accordingly, R.S. 400 is independent of personal computer type and the operating system 428 it may be using. With reference to FIG. 2, to access network 10, a user preferably has a personal computer 405 with at least 512K RAM and a single disk drive 416. The user typically accesses network 10 using a 1,200 or 2,400 bps modem (not shown). To initiate a session with network 10, objects representing the logon application are retrieved from the user's personal diskette, including the R.S. application software, which was previously set up during a standard installation - enrollment procedure (not explained further). Once communication between RS 400 and cache concentrator layer 300 has been established, the user begins a standard logon procedure by inputting a personal entry code. Once the logon procedure is complete, the user can begin to access various desired services (i.e., partitioned applications) providing the desired information display and/or transaction operations.

APPLICATIONS AND PAGES

Applications, i.e. information events, are composed of a sequence of one or more pages opened at screen 414 of monitor 412. This is better seen with reference to FIG 3 (a) and 3(b) were a page 255 is illustrated as might appear at screen 414 of monitor 412. With reference to FIG 3(a), each page 255 is formatted into page partitions 250, 260, 280, and 290 (not to be confused with applications partitions). Window page partitions 275 well known in the art are also available and are opened and closed conditionally on page 255 upon the occurrence of an event specified in the application being run each 250-290 and window 275 is made up of a page element which define the content of the partition or window.

Each page 255 includes: a header page partition 250, which has a page element associated with it and which typically conveys information on the page's topic or sponsor; one or more body page partitions 260 and window page partitions 275, each of which is associated with a page element which as noted gives providing the information and transactional content of the page. For example, a page element may contain presentation data selected as a menu option in the previous page, and/or may contain prompts to which a user responds in pre-defined fields to execute transactions. As illustrated in FIG. 3(b), the page element associated with body page partition 260 includes display fields 270. A window page partition 275 represents the same informational and transactional capability as a body partition, except greater flexibility is provided for its location and size.

Continuing with reference to FIG. 3(b), advertisements 280 provided over network 10 like a page elements also include information displayed on page 255 may be included in any partition of a page. Advertisements 255 may be presented to the user on an individual basis from queues of advertisements constructed off-line by business system 130, and sent to file server 205 where they are accessible to each RS 400.

Individual queues of advertisements are constructed based upon data collected on the partitioned applications that were accessed by a user, and upon events the user generated in response to applications. The data are collected and reported by RS 400 to a data collection co-application in file-server 205 for later transmission to business system 130. In addition to application access and use characteristics, a variety of other parameters, such as user demographics or postal ZIP code, may be used as targeting criteria. From such data, queues of advertisements are constructed and targeted to either individual users or to sets of users who fall into certain groups according such parameters.

Also with reference to FIG. 3b, a user interface 285 is displayed on the page, enabling the user to interact with the network RS 400 and the other elements of network 10 to cause such operations as navigating from page to page, performing a transaction, or obtaining more information about other applications. As shown in FIG 3b, user interface 285 includes a command

r 290 having a number of commands 291-295 which the user can execute. The functions of commands 291-298 are discussed in greater detail below. interface, are discussed in greater detail below.

NETWORK OBJECTS

As noted above, in conventional time-sharing computer networks, the data and program instructions necessary to support user sessions are maintained at a central host computer. However, that approach has been found to create processing bottlenecks as greater numbers of users are connected to the network; bottlenecks which require increases in processing power and complexity; e.g., multiple hosts of greater computing capability, if the network is to meet demand. Further, such bottlenecks have been found to also slow response time as more users are connected to the network and seek to have their requests for data processing answered.

The consequences of the host processing bottlenecking is to either compel capital expenditures to expand host processing capability, or accept longer response times; i.e., a slower network, and risk user dissatisfaction.

However, even in the case where additional computing power is added, and where response time is allowed to increase, eventually

the host becomes user saturated as more and more users are sought to be served by the network.

The method and apparatus of this invention are directed at alleviating the effects of host-centered limitations, and extending the network saturation point. In accordance with the invention, this is achieved by reducing the demand on the host for processing resources by structuring the network so that the higher network levels act primarily to maintain and supply data and programs to the lower levels of the network, particularly RS 400, which acts to manage and sustain the user screen displays.

More particularly, the method aspect of the invention features procedures for parsing the network data and program instructions required to support the interactive user sessions into packets, referred to as objects, and distributing them into the network where they can be processed at lower levels, particularly, reception system 400.

In accordance with the invention, the screens presented at the user's monitor are each divided into addressable partitions shown in FIG. 3a, and the display text and graphics necessary to make up the partitions, as well as the program instructions and control data necessary to deliver and sustain the screens and partitions are formulated from precreated objects. Further, the objects are structured in accordance with an architecture that permits the displayed data to be relocatable on the screen, and

be reusable to make up other screens and other sessions, either as precreated and stored sessions or interactive sessions, dynamically created in response to the user's requests.

In accordance with the method aspect of the invention and as shown in FIG.4c, the network objects are organized as a family of objects each of which perform a specific function in support of the interactive session. More particularly, the network object family is seen to include 6 members: page format objects 502, page element object 504, window objects 506, program objects 508, advertisement objects 510 and page template objects 500.

Within this family, page format objects 502 are designed to define the partitioning 250 to 290 of the monitor screen shown in Fig. 3a. The page format objects 502 provide a means for pre-defining screen partitions and for ensuring a uniform look to the page presented on the reception system monitor. They provide the origin; i.e., drawing points, and dimensions of each page partition and different values for presentation commands such as palette and background color.

Page format objects 502 are referenced whenever non-window data is to be displayed and as noted ensure a consistent presentation of the page. In addition, page format objects 502 assures proper tessellation or "tiling" of the displayed partitions.

Page element objects 504, on the other hand, are structured to contain the display data; i.e., text and graphic, to be displayed which is mapped within screen partitions 250 to 290, and to further provide the associated control data and programs. More specifically, the display data is described within the object as NAPLPS data, and includes, PDI, ASCII, Incremental Point and other display encoding schemes. Page element objects also control the functionality within the screen partition by means of field definition segments 516 and program call segments 532, as further described in connection with the description of such segments. Page element objects 504 are relocatable and may be reused by many pages. To enable the displayable data to be re-located, display data must be created by producers in the NAPLPS relative mode.

Continuing with reference to FIG.4c, window objects 508 include the display and control data necessary to support window partitions 275 best seen in FIG.3a. Windows contain display data which overlays the base page and control data which supersede the base page control data for the underlying screen during the duration of the window. Window objects 506 contain data which is to be displayed or otherwise presented to the viewer which is relatively independent from the rest of the page. Display data within windows overlays the base page until the window is closed. Logic associated with the window supersedes base page logic for the duration of the window. When a window is opened, the bit-map of the area covered by window is saved and most logic functions

for the overlaid page are de-activated. When the window is closed, the saved bit-map is swapped onto the screen, the logic functions associated with the window are disabled, and prior logic functions are reactivated.

Windows are opened by user or program control. They do not form part of the base page. Windows would typically be opened as a result of the completion of events specified in program call segments 532.

Window objects 506 are very similar in structure to page element objects 504. The critical difference is that window objects 506 specify their own size and absolute screen location by means of a partition definition segment 528.

Program object 508 contain program instructions written in a high-level language called TRINTEX Basic Object Language, i.e., TBOL, described in greater detail hereafter, which may be executed on reception system 400 to support the application. More particularly, program objects 508 includes interpretable program code, executable machine code and parameters to be acted upon in conjunction with the presentation of text and graphics to the reception system monitors.

Program objects 508 may be called for execution by means of program call segments 532, which specify when a program is to be

executed (event), what program to execute (program pointer), and how programs should run (parameters).

Programs are treated as objects to conform to the open-ended design philosophy of the DOA, allowing the dissemination of newly developed developed programs to be easily and economically performed. It is desirable to have as many of these program objects staged at or close to RS 400 as possible.

Still further, advertising objects 510 include the text and graphics that may be presented at ad partition 280 presented on the monitor screen as shown in FIG.3b.

Finally, the object family includes page template objects 500. Page template objects 500 are designed to define the components of the full screen presented to the viewer. Particularly, page template objects 500 include the entry point to a screen, the name of the page format objects which specify the various partitions a screen will have and the page element object that contain the display data and partitioning parameters for the page.

Additionally, page template object 500 includes the specific program calls required to execute the screens associated with the application being presented to the user, and may serve as the means for the user to selectively move through; i.e., navigate the pages of interest which are associated with various

applications. Thus, in effect, page template objects 500 constitute the "recipe" for making up the collection of text and graphic information required to make the screens to be presented to the user.

Also in accordance with the invention, object 500 to 510 shown in FIG.4c are themselves made up of further sub-blocks of information that may be selectively collected to define the objects and resulting pages that ultimately constitute the application presented to the user in an interactive text and graphic session.

More specifically and as shown schematically in FIG.4a, objects 500 to 510 are predefined, variable length records consisting of a fixed length header 551 and one or more self-defining record segments 552 a list of which is presented in FIG.4c as segment types 512 to 541.

In accordance with the invention, and as shown in FIG. 4b, object header 551 in preferred form is 18 bytes in length and contains a prescribed sequence of information which provides data regarding the object's identification, its anticipated use, association to other objects, its length and its version and currency.

More particularly, each of the 18 bytes of object header 551 are conventional hexadecimal, 8 bit bytes and are arranged in a

six pattern to facilitate interpretation by network 10. Particularly, and as shown in FIG.4b, the first byte of header 551; i.e., byte 1, identifies the length of the object ID in hexadecimal. The next six bytes; i.e., bytes 2 to 7, are allocated for identifying access control to the object so as to allow creation of closed user groups to whom the object(s) is to be provided. As will be appreciated by those skilled in the art, the ability to earmark objects in anticipation of user requests enables the network anticipate requests and pre-collect objects from large numbers of them maintained to render the network more efficient and reduce response time. The following 4 bytes of header 551; bytes 8 to 11, are used to identify the set of objects to which the subject object belongs. In this regard, it will be appreciated that, again, for speed of access and efficiency of selection, the objects are arranged in groups or sets which are likely to be presented to user sequentially in presenting the page sets; i.e., screens that go to make up a session.

Following identification of the object set, the next byte in header 551; i.e., byte 12, gives the location of the subject object in the set. As will be appreciated here also the identification is provided to facilitates ease of object location and access among the many thousands of objects that are maintained to, thereby, render their selection and presentation more efficient and speedy.

Thereafter, the following bytes of header 551; i.e., byte 13, designates the object type; e.g., page format, page template, page element, etc. Following identification of the object type, two bytes; i.e., bytes 14, 15, are allocated to define the length of the object, which may be of what ever length is necessary to supply the data necessary, and thereby provides great flexibility for creation of the screens. Thereafter, a single byte; i.e., byte 16, is allocated to identify the storage characteristic for the object; i.e., the criterion which establishes at what level in network 10 the object will be stored, and the basis upon which it will be updated. At least a portion of this byte; i.e, the higher order nibble (first 4 bits reading from left to right) is associated with the last byte; i.e., byte 18, in the header which identifies the version of the object, a control used in determining how often in a predetermined period of time the object will be updated by the network.

Following storage characteristic byte 16, header 551 includes a byte; i.e., 17, which identifies the number of objects in the set to which the subject object belongs. Finally, and as noted above, header 551 includes a byte; i.e., 18, which identifies the version of the object. Particularly the object version is a number to establish the control for the update of the object that are resident at reception system 400.

As shown in FIG.4a, and as noted above, in addition to header 551, the object includes one more of the various segment types shown in FIG.4c.

Segments 512 to 541 are the basic building blocks of the objects. And, as in the case of the object, the segments are also self-defining. As will be appreciated by those skilled in the art, by making the segments self-defining, changes in the objects and their use in the network can be made without changing preexisting objects.

As in the case the objects, the segments have also been provided with a specific structure. Particularly, and as shown in FIG. 4a, segments 552 consists of a designation of segment type 553, identification of segment length 554, followed by the information necessary to implement the segment and its associated object 555; e.g., either, control data, display data or program code.

In this structure, segment type 553 is identified with a one-byte hexadecimal code which describes the general function of the segment. Thereafter, segment length 554 is identified as a fixed two-byte long field which carries the segment length as a hexadecimal number in INTEL format; i.e., least significant byte first. Finally, data within segments may be identified either by position or key word, depending on the specific requirements of the segment.

In this structure, segment type 553 is identified with a one-byte hexadecimal code which describes the general function of the segment. Thereafter, segment length 554 is identified as a fixed two-byte long field which carries the segment length as a hexadecimal number in INTEL format; i.e., least significant byte first. Finally, data within segments may be identified either by position or keyword, depending on the specific requirements of the segment.

In accordance with the invention, the specific structure for the objects and segments shown in FIG. 4c would be as described below. Object and segment structure is described, and segment function is briefly described. [ON NOTATION: the following description of the structure for objects and segments uses the following key conventions]:

< > mandatory item

( ) optional item

... item may be repeated

```
|item| |item|   items in a column indicate either/or
<    > (    )
|item| |item|
```

OBJECTS

PAGE TEMPLATE OBJECT:

<header>

<page format call>

(program call)...

(system table call)...

(keyword/navigation)...

(compression descriptor)

(page element call)...

(page element selector)

(external reference)

PAGE FORMAT OBJECT:

<header>

(page defaults)

(compression descriptor)

<partition definition>

PAGE ELEMENT OBJECT:

<header>

(presentation data)...

(custom cursor)...

(field definition)...

(custom cursor type 2)..

(field definition type 2)...

(inventory control)

(compression descriptor)

(program call)...

(custom text)...

(field level program call)...

(custom graphic)...

(array definition)...

WINDOW OBJECT:

<header> (compression description) <partition definition> (page element call) (presentation data)... (program call)... (custom cursor)... (custom text)... (custom cursor type 2)... (custom graphic) (field definition)... (field level program call)... (field definition type 2)... (array definition)... (inventory control)

PROGRAM OBJECTS:

<header> (compression descriptor) <program data> ...

SEGMENTS

PROGRAM CALL SEGMENT

Program call segments 532 are used to invoke programs. Program events will be specified in logical terms and will be mapped by the reception system native software 420 to specific physical triggers (e.g., the "logical" event end-of-page may map to the physical <ENTER> key). The logical event to be completed to initiate the program is specified in a one-byte token within the

segment. The structure of program call segment 532 is as follows:

```
                               |prgm obj. id|
<st> <sl> <event> <prefix> <                  > (parm)...
                               |displacement|
```

where "event" is a one-byte token of the logical event to be completed to initiate the program; "prefix" is a one-byte prefix to an object id or displacement; "object id" is id of the program object 506; "displacement" is a pointer to an imbedded program call segment 532; and "parm" is the parameters specific to the program.

FIELD LEVEL PROGRAM CALL SEGMENTS

Some programs, such as edits, must be triggered at the field level. Field-level program call segments 518 relate program calls to specified field definition segments 516. The structure of field-level program call segments is as follows:

```
                                          |prgm.obj.id|
<st> <sl> <event> <field id> <prefix><                  > (parm)...
                                          |displacment|
```

where "event" is a one-byte token of the logical event to be completed to initiate the program; "field id" is the one-byte name of the field specified in a field definition segment 516

with which this call segment is associated; "prefix" is a one-byte prefix to an object id or displacement; "object id" is id of the program object 506; "displacement" is a pointer to an imbedded program call segment 532; and "parm" is the parameters specific to the program.

PROGRAM DATA SEGMENT

Program data segments 536 contain the actual program data to be processed by RS 400. Program data may include either source code, compiled machine code, macros, storage maps, and/or parameters. The structure of program data segments 536 is as follows:

<st> <sl> <type> <program data>

where "type" refers to the type of program data contained (1=TBOL, 2=table data).

COMPRESSION DESCRIPTOR SEGMENT

Compression descriptor segment contains information need for the decompression of objects compressed in interactive network 10. The segment is a formalization of parameters to be used by a decompression routine residing at the RS 400, using Huffman encoding well known the art. The structure of compression descriptor segment 538 is:

<st> <sl> <table number> <length 1> (length 2)

Where "table number" is a one-byte number corresponding to the 'class' indicator in the table structure segment ____ of the appropriate decompression system table object; "length 1" is a two-byte indicator of the length of the segment after compression (not including object header and length of compression descriptor); and "length 2" is a two-byte indicator of the length of the segment before compression (not including object header and length of compression descriptor).

PAGE DEFAULT SEGMENTS

Page default segments 540 specify defaults for the entire page using NAPLPS commands. The structure of page default segment 540 is:

<st> <sl> <NAPLPS>

PARTITION DEFINITION SEGMENT

Partition definition segment 528 describes display screen areas into which data may be mapped. The structure of partition definition segment 528 is:

<st> <sl> <partition id> <origin> <size> (NAPLPS)

where "partition id" is a one-byte partition id unique within the current page format object 502; "origin" is the partition origin point, a three-byte NAPLPS point set (absolute, invisible) operand contained the absolute coordinates of the lower left

corner of the partition; and "size" refers to partition size, a three-byte NAPLPS point set (absolute, invisible) operand containing the absolute coordinates of the upper right corner of the partition.

PAGE FORMAT CALL SEGMENT

Page format call segment 526 is used by the page template object 500 to specify the particular page format object 502 to be used as the "blueprint" of the page. Page format call segment 526 structure is as follows:

<st> <sl> <prefix> <object id>

where "prefix" is a one-byte prefix to an object id or displacement; and "object id" is the object id of the page format object 502.

PAGE ELEMENT CALL SEGMENT

Page element call segment 522 specifies which data is to be present on the base page and in which page partition the data is to appear. The structure of page element call segment is as follows:

```
                                            |object id|
<st> <sl> <partition id> <priority> <prefix> <          >
                                            |displcmnt|
```

where "partition id" is the partition id, as specified in the page format object 502 upon which this object will act; "priority" is a one-byte binary flag indicating priority (from 0-15 with 0 indicating no priority [FIFO}) of object interpretation (high-order nibble) and of painting (low-order nibble); "prefix" is a one-byte object id or displacement; "object id" is the id of the page element object 522; and "displacement" is a pointer to an imbedded page element object 522.

PAGE ELEMENT SELECTOR SEGMENT

Page element selector segment 524 provides a mechanism by which page elements may be dynamically selected for presentation within a partition. The structure of page element selector segment 524 is:

```
                                               |pgm.obj.id|
<st> <sl> <part.id> <priority> <prefix>   <               >(parm)..
                                               |displacement|
```

where "part. id" is the partition id as specified within the page format object 502 upon which the object will act; "priority" is a one-byte binary flag indicating priority (from 0-15 with 0 indicating no priority [FIFO}) of object interpretation (high-order nibble) and of painting (low-order nibble); "prefix" is a one-byte object id or displacement; "pgm.obj.id" is the object id of the program object 508 used to dynamically select an element object; "displacement" is a pointer to an imbedded program object

508, and "parm" is parameters which are used by the program object 508.

SYSTEM TABLE CALL SEGMENT

System table call segments 542 call system table segments for use by the RS 400. Each table entry in a system table segment contains an index-addressable segment (e.g., a set of custom text segments 514). System table call segments operate in a "locked-shift" mode, meaning that each system table of a particular class will remain operative until a new table is requested for that class of table. System table call segment 542 structure is as follows:

```
                    |object id|
<st> <sl> <prefix> <           >
                    |displcmnt|
```

where "prefix" is a one-byte prefix to an object id or displacement; "object id" is the id of a system table segment; and "displacement" is a pointer to an inbedded system table segment.

TABLE STRUCTURE SEGMENT

Table structure segments 554 describe the basic class and composition of system table objects. The structure of table structure segment 554 is:

```
<st> <sl> <class> <number of entries> <maximum entry length>
```

Where "class" is a one-byte identifier indicating the class of the current table (as follows:

x'00' = custom text table

x'01' = custom cursor table

x'02' = custom graphic table

x'03' = custom cursor type 2 table

x'30' thru x'39' = decompression table);

"number of entries is a two-byte field specifying the total number of entries contained in the current table; and "maximum entry length" is a two-byte field specifying the length of the largest entry in the current table.

TABLE ENTRY SEGMENT

Table entry segment 556 contains the actual data that has been placed in tabular form. The meaning of the data is derived from the class indicator in the table structure segment 554. They will be treated as functional equivalent of certain other segments such as custom text segment 514 or custom cursor segment 512. Table entry segment structure is:

<st> <sl> <data>

where "data" is the data contained in the entry (text character attributes if table belongs to the custom text class; NAPLPS if the table belongs to the custom cursor class).

EXTERNAL REFERENCE SEGMENT

External reference segment 544 is provided to improve run-time performance by providing the RS 400 with a list of objects that are candidates for pre-fetching. External reference segments 544 contain a list of object ids which are used within the current page. Each object indicated within this list is called explicitly from the current frame. Object ids specified within the external reference segment 544 will take advantage of the notion of "inheritance." If multiple object ids are contained within the segment, they may inherit high-order bytes from previously specified ids, thus avoiding repetition of information that is inherited (e.g. to specify objects ABC12, ABC22, and ABC37 in this segment, one encodes them as ABC12, 22, 37). External reference segments 544 operate in a "locked-shift" mode, meaning that each external reference list will be active until the next external reference list is encountered. In the best mode, there should be no more than one external reference segment per page. External reference segment structure is as follows:

<st> <sl> <# of ids> <priority> <prefix> <object id>

where "# of ids" is a one-byte field specifying the total number of object ids contained in the current segment; "priority" is a one-byte priority value specifying priority of pre-fetch (priorities may be duplicated, in which case they will be processed from left to right); "prefix" is a one-byte prefix to an object id or

displacement; and "object id" is the id of an externally referenced object.

KEYWORD/NAVIGATION SEGMENT

Keyword/navigation segments 520 may contain two types of information: (1) references to other page template objects 500 that are either logically higher than the current page template (e.g., a "parent" menu) or references to page template objects 500 outside the current "world" (a logically cohesive group of pages having a single entry point, such as a general map of the interactive service); or (2) a character string to be associated with the current page template object 500, which may be displayed to the user to indicate an alternative path or keyword which could be used to access the current page template. The structure of keyword/navigation segment is as follows:

<st> <sl> <#ids> (<prefix> <object id>)...(character string)

where "#ids" is the number of object ids in this segment; "prefix" is a one-byte object id prefix; "object id" is an object id associate witha the current page as either an upward hierarchical reference or a non-hierarchical reference; and "character string" is the character string to be associated with the current page. (See also, discussion of JUMPword navigation, below).

PRESENTATION DATA SEGMENT

Presentation data segments 530 contain the actual contain the actual data to be displayed or otherwise presented to the user. Presentation data may contain NAPLPS codes, ASCII, and other codes for visual display. Presentation data may in the future contain codes for the presentation of audio signals. The structure of presentation data segment is:

<st> <sl> <type> <size> <presentation data>

where "type" is the type of presentation data included in this segment (1=NAPLPS, 2=ASCII); "size" is a NAPLPS operand that defines the upper right portion of the display data; and "presentation data" is the actual data to be presented to the user.

FIELD DEFINITION SEGMENT

Field definition segments 516 define the location of a field, name the field, and specify how data will be acted on within the named field. Field definition segment 516 structure is as follows:

<st> <sl> <attributes> <origin> <size> <name>
<text id> (cursor id) (cursor origin)

where the structure is defined as below.

"Attributes" of a field define ways in which the user interacts with RS 400 at a rudimentary level. Three basic field types are

supported: (1) unprotected fields into which users may enter data; (2) protected fields into which users may position the cursor, function and enter keys, but may not enter data; and (3) skip fields which are inaccessible to the user keyboard. Additional attributes which may be specified for a field include: numeric input only (unprotected); alphabetic input only (unprotected); foreground color; and background color. Attributes are encoded in two bytes. The first nibble of the first byte is a hexadecimal number (O - F) that represents the foreground color selection from the in-use palette. The second nibble of the first byte is a hexadecimal number (O - F) that represents the background color selection from the in-use palette. The first nibble of the second byte consists of a set of bit flags which, from left to right, indicate:

bit 0 if '1' : protect on
bit 1 if '1' : automatic skip on
bit 2 if '1' : numeric input only
bit 3 if '1' : alphabetic input only.

The second nibble of the second byte is reserved for future expansion.

"Origin" is a three-byte NAPLPS point set (relative, invisible) operand that defines the lower left corner of the field.

"Size" is a three-byte NAPLPS point set (relative, invisible) operand that defines the upper right corner of the field.

"Name" is a one-byte name assigned to the field so that it may be accessible to programs.

"Text id" is a one-byte id of the text characteristics to be associated with the field (e.g., size, gapping, proportional spacing, etc.).

"Cursor id" is a one-byte id of the cursor type to be associated with the field.

"Cursor origin" is a three-byte NAPLPS operand specifying relative draw point to the cursor. If this operand is not present, the cursor origin point will be assumedn to be the same as the field origin point.

FIELD DEFINITION TYPE 2 SEGMENT Field definition type 2 segments 548 are provided to enhance run-time flexibility of fields. Field definition type 2 segment structure is as follows:

<st> <sl> <attributes> <origin> <size> <name>
<text id> <cc ll> (<cursor id> (cursor origin))
<# hot spots> (<hs ll> <hssize> (hsorigin))...
(<cg ll> <cgraphic id> <cgmode> (cgorigin))...

where structure is defined below.

"Attributes" of a field describe how the user and RS 400 interact at a rudimentary level. Attributes for field definition type 2 segments 548 are contained in four bytes:

```
Byte 1..............Field type
  bit 0..............TBOL interpreter indicator
                     '0' no fire
                     '0' fire
  bits 1-7...........Interaction type
                     x'000 0000 : input (unprotected)
                     x'000 0001 : action (protected)
                     x'000 0010 : display (askip)
                     x'000 0100 : hidden (dark) Byte Byte
2..............Text Attributes (bit flags)
  bits 0-7...........x'0000 0001 : left justify
                     x'0000 0010 : right justify
                     x'0000 0100 : word wrap Byte Byte
3..............Data Type
  bits 0-7...........x'0000 0001 : alphabetic
                     x'0000 0010 : numeric
                     x'0000 0100 : password Byte Byte
4..............Color
  bits 0-3...........foreground color
  bits 4-7...........background color
```

"Origin" is a three-byte NAPLPS point set (relative, invisible) operand that defines the lower left corner of the field.

"Size is a three-byte NAPLPS point set (relative, invisible) operand that defines the upper right corner of the field.

"Name" is a one-byte name assigned to the field so that it maybe accessible to the program.

"Text id" is a one-byte id of the text characteristics to be associated with the field, such as size, gapping, proportional, etc.

"cc ll" is the cursor length; a one-byte field describing the combined length of the cursor id field and the cursor origin field. If the length contains a 1, then the cursor origin operand is not present, in which case, the cursor origin defaults to the field origin point.

"Cursor id" is a one-byte id of the cursor type to be associated with the field.

"Cursor origin" is a three-byte NAPLPS operand specifying the relative draw point of the cursor. If this operand is not present, the cursor origin point will be assumed to be the same as the field origin point.

"# hot spots" is the number of hot spots used by this field. "Hot spots" fefers to a set of coordinates that will be selectable by a pointing device, such as a mouse. If the contents of this field are zero, the hot spot for the field will be assumed to be the coordinates that are covered by the custom cursor.

"Hot spot sets" facilitate assigning a variable number of hot spots to a field. Each hot spot is described by a set of operands consisting of hot spot length, origin, and size. Each set of such operands describes one hot spot. When using multiple hot spots, multiple sets of operands must be present.

"hs ll" or hot spot length is a one-byte binary field describing the length of the hot spot coordinates for a hot spot "instance." If this byte contains zero, the hot spot origin and size default to the coordinates described by the custom cursor. If this byte contains 3, then the hot spot origin point will not follow, but will default to the custom cursor origin point. If this byte contains 6, then both the hot spot origin and size are present.

"Hot spot size" is a three-byte NAPLPS x,y coordinate describing the top right corner of the hot spot.

"Hot spot origin" is a three-byte NAPLPS x,y coordinate describing the lower left corner of the hot spot. If the hot spot lenth is equal to 3, this field is not present. In that case, the hot spot origin point defaults to the origin point of the custom

cursor (which may have also defaulted to the field origin point). If the hot spot lenth is equal to 6, then this field is present.

A custom graphic operand set contains four operands:

"cg ll" or custom graphic set length, which, if 2, then no custom graphic origin is present. In that case, the origin point of the custom graphic defaults to the field origin point;

"cg id" or custom graphic id, a one-byte identifier of a custom graphic string;

"cgmode" or custom graphic mode, which is one byte used to describe variable conditions that apply to the graphic. Defined values include:

x'01 : blink

x'02 : dynamic

x'03 : permanent

"cgorigin" or custom graphic origin, a three-byte NAPLPS x,y coordinate indicating the lower left corner of the custom graphic. If this operand is not present, the lower left corner will default to the field origin point.

ARRAY DEFINITION SEGMENT

Array definition segments 548 define the names and relative locations of fields in a row that makes up an array or table. The first row of fields must have been defined using field definition segments 516. The array definition provides a shorthand for specifying the replication of selected fields from the initial page. The structure of the array definition segment 548 is as follows:

<st> <sl> <#occurrences> <vertical gap> <field name> ...

where "#occurrences" is a one-byte field describing the number of rows to be generated to create the array (the first row is assumed to be generated from field definition segments 516); "vertical gap" is a NAPLPS point set operand (relative, invisible) containing the DY of inter-row spacing; and "field name" is a one-byte name (from the field definition) of the fields in a row of the array. **CUSTOM GRAPHICS SEGMENT**

Custom graphics segment 550 provides a means to package graphics commands. These graphics commands may be related to a field and initiated based on run-time conditions. The structure of custom graphics segment 550 is as follows:

<st> <sl> <id> <size> <NAPLPS>

where "id" is a one-byte identifier for this custom graphic; "size" is a three-byte NAPLPS operand specifying upper right

corner of the graphic area in a relative mode; and "NAPLPS" are NAPLPS commands to paint the custom image.

CUSTOM CURSOR SEGMENT

Custom cursor segment 512 allows fancy graphics to be associated with cursor positioning in a field. Using this segment, cursors may be defined to any size or shape and may be placed at any desired location relative to their associated fields. The structure of custom cursor segment 512 is as follows:

<st> <sl> <id> <size> <NAPLPS>

where "id" is a one-byte identifier for this custom cursor; "size" is a three-byte NAPLPS operand specifying upper right corner of the cursor area in a relative mode; and "NAPLPS" are NAPLPS commands to paint the custom image.

CUSTOM CURSOR TYPE 2

Custom cursor type 2 segment 552 allows cursors to be defined to any size or shape and may be placed at any desired location relative to their associated fields. The structure of custom cursor type 2 segment 552 is as follows:

<st> <sl> <id> <size> (<ll> <NAPLPS>)...

where "id" is a one-byte identifier for this custom cursor; "size" is a three-byte NAPLPS operand specifying upper right

corner of the cursor area in a relative mode; "11" is the length of the following NAPLPS data; and "NAPLPS" are NAPLPS commands to paint the custom image.

CUSTOM TEXT SEGMENT

Custom text segments 514 allow the definition of custom display of text within a field when non-standard character field size is used (20 x 40 display characters is standard) or custom spacing, movement, or rotation of characters is desired. The structure of custom text segments 514 is as follows:

<st> <sl> <id> <NAPLPS>

where "id" is a one-byte identifier for this TXT command; and "NAPLPS" are NAPLPS commands specifying character field size, rotation, movement, inter-row and inter-character text gaps.

INVENTORY CONTROL SEGMENT

Inventory control segment is provided to facilitate management of objects. The inventory segment is structured:

<st> <sl> <type> <inventory number> (sub-number)

where type is a one-byte indicator showing object usage (as follows:

0 = no defined use

1 = leader ad

2 = ad campaign completion

3 = leader ad completion

4 - 255 = reserved for future use);

"inventory number" is a unique two-byte number to be used for inventory control and statistics; and "sub-number is the same as inventory number.

At Page 27, Line 24, is inserted the following:

NETWORK MESSAGES

In addition to the network objects and the display and control data, and the program instructions previously they contain as described, network 10 also exchanges information regarding the support of user sessions and the maintenance of the network as "messenger". Specifically, the messages typically relates to the exchange of information associated with initial logon of a reception system 400 to network 10, dialogue between reception system 400 and other elements and communications by the other elements amongst themselves.

In accordance with the invention, to facilitate message exchange internally, and through gateway 210 to entities externally to network 10, a protocol termed the "Data Interchange Architecture" (DIA) is used to support the transport and

interpretation of information. More particularly, DIA enables: communications between RS 400 units, separation of functions between network layers 100, 200, 300 and 401; consistent parsing of data; an "open" architecture for network 10; downward compatibility within the network, compatibility with standard industry protocols such as the IBM System Network Architecture and Open Systems. Interconnections Standard; support of network utility sessions and standardization of common network and application return codes.

Thus DIA binds the various components of network 10 into a coherent entity by providing a common data stream for communications management purposes. DIA provides the ability to route messages between applications based in IBM System Network Architecture (SNA), (well known in the art, and more fully described in Data and Computer Communications, by W. Stallings, Chapter 12, McMillian Publishing, Inc. (1985)) and non-SNA reception system applications; e.g. home computer applications. Further, DIA provides common data structure between applications run at RS 400 units and applications that may be run on external computer networks; e.g. Dow Jones Services, accessed through gateway 210. As well, DIA provides support for utility sessions between backbone applications run within network 10.

In make up, DIA is a blend of SNA and Non-SNA based modes, and thus provides a means for combining the differences between these modes within network 10. Accordingly, the action of DIA differs

Depending on whether DIA is operating within an SNA portion of network 10 or whether it is operating within the non-SNA portion of the network. More specifically, within the SNA portion of network 10, DIA and its supporting programs may be considered "applications" facilities. In this context, DIA resides at the transaction services level of SNA, also known as the Specific Application level of Open Systems Interconnections (also discussed in chapter 12 of Data and Computer Communications by W. Stallings above noted). However, in either case, it is a level 7 facility.

Within non-SNA portions of network 10, DIA and its supporting programs provide routing, transport, sessions, and some transaction facilities. Thus DIA provides a comprehensive network architecture providing OSI level 3, 4, 5 and 7 services.

In accordance with the invention, DIA facilitates "utility session" within network 10. Utility sessions allow partner applications to communicate by means of the single session established between two logical units of the SNA type. In order to reduce the number of resources which must be defined to the network support programs, many user messages may be passed to many different application destinations through logical unit to logical unit (LU-LU) "pipes".

Applications exist on either side of the LU-LU pipe which act to concentrate outbound messages en route to applications

esident on the other side of the LU-LU pipe; distribute inbound messages to local applications; and maintain and manage application task boundaries. Users may enter into a conversation with a set of transactions, refined to tasks, which are hereafter noted as "user sessions", and the boundaries of these user sessions (tasks) are indicated by begin session/end session flags.

Another application function supported by DIA is the routing of messages between nodes of network 10. Particularly, a switching application will route messages to the appropriate LU-LU session for transmission to another mode by examining and resolving the DIA destination IDs hereafter described.

In accordance with the invention messages conforming to DIA are composed of two functional parts: message headers and message text. Message Headers are transparent to most applications, but are the primary vehicle for passing information for session-layer to session-layer or transport-layer to transport-layer communications. Further, Message Text which is processed by end-users, and is transparent to session and transport mechanisms.

In order to reduce program complexity and facilitate maintenance and enhancements, DIA has been structured in a layered fashion. In this regard, the DIA-defined data which flows through network 10 consists of a set a headers preface the end-user to end-user message text. Further, as in the case of objects, messages are organized in a family of types based on the

specific form of its header. Particularly, there are "FMO" headers which contain routing and control information; FM2 headers which contain transport level information; FM4 headers which contain gateway information; FM8 headers which obtain information for secondary routing; i.e. messages passed through from node to node; FM9 headers which contain network management information; and FM64 headers which contain application-to-application management information, where, for example, applications running at RS 400 need be rendered compatible with applications running on an external computer connected to network 10 through a gateway 210.

In order to provide SNA compatibility, the first two bytes of all DIA FM headers are formatted such that byte 1 defines the length of header in hexadecimal; and byte 2, bit 0, identifies whether concatenation is provided or not; e.g. if bit 1 = 0 no other headers follow, but if bit 1 = 1, then the current header is followed by a concatenated header; while bits 1 -7 identify the header type in hexadecimal value.

As will be appreciated to those skilled in the art, this layout is the same as that of SNA Function Management Headers. In an SNA LU0 implementation the DIA FM headers may be treated as SNA Function Management Headers (FMHs). Alternatively, the DIA FMs may be treated as pure data within the SNA Request Unit (RU).

With regard to destination routing, the basic premise of DIA is that each message flowing through network 10 carries a DIA header (FMO) that identifies its source and destination ids. Accordingly, switching applications exist which map destination ids to resources and route messages appropriately. In accordance with the invention, in order to send a reply, the recipient application simply swaps the content of the destination and source id fields and return message.

In the context of DIA the totality of ports, devices, and programs which are managed by a particular Switch and defined as destinations, are referred to as "regions". In this regard, each Switch; i.e. server 205 or cache/concentrator 302 shown in Fig. 2, need only be aware of the destination ids of resources within its own region and of the destination ids of switches resident in immediately adjacent nodes. Since server 205 is the central hub within the network 10 for application message routing, messages destined for end-users unknown to a switch are routed toward server 205 for eventual resolution. Destination id naming conventions then enable server 205 to determine the appropriate switch to which the message should be forwarded. Particularly, "destination id" fields "regions" and "unit" are used for this purpose.

Concerning switch responsibility, a switching application has three primary responsibilities. It must forward messages to adjacent switches. It must collect messages from, and distribute

messages to resources within its own region. And, it must maintain and manage application task boundaries. Users may enter into a conversation with a set of transactions. This set of transactions is referred to as a "task". These tasks are called user sessions. Further, the boundaries of these tasks are indicated by begin session/end session flags.

In order to fulfill these functions, a resource definition facility must exist for each switch to map each addressable resource to a destination id. In some cases, particularly on the RS 400, it may be desireable for an application to dynamically define subordinate resources to the switch and to interact with the switch to generate unique destination ids for these subordinate resources. It may also be necessary for the switch to either communicate with, or act within an application subsystem. An example of an application subsystem is the Customer Information Control System, (CICS) event, where CICS is a commercially available transaction process controller of the IBM Company, well known in the art. CICS, although subordinate to the operating system, is responsible for initiating and managing application "transaction" programs. Routing to specific transactions under the control of an application subsystem may be accomplished by a secondary address. In this case, the subsystem is defined as the primary destination. The transaction is defined as the secondary destination. A switch must only route incoming messages to the subsystem. The subsystem in turn posts to, or initiates the desired transaction.

The use of secondary addressing provides several advantages. Particularly, switch resource tables are not affected by the coming and going of "transaction" applications. Further, since the DIA headers are SNA compatible, Type 1 application such as CICS need have no special message routing functions. A switch configured in accordance with the IBM standard VTAM could route incoming messages to CICS. Still further, transactions need not go into "receive loops". It is possible for the subsystem to poll on behalf of many transaction programs. In accordance with DIA, secondary addressing is implemented within the application data stream. For instance, CICS transaction ids are, by convention, to be found in the first four bytes of application text.

With regard to the standards for DIA, it will be recalled that DIA messages have a header followed by the message information. In the preferred embodiment, the DIA headers may be concatenated to one another. Further, the presence of concatenated headers is indicated by the setting of the first bit (bit 0) of the Header Type field.

However, there are two restrictions on the use of concatenated headers. Particularly, concatenated headers are required to be sequenced in ascending order left to right by header type numbers and secondary message text prefaced by concatenated headers (such as FM64 architectured message text) are not permitted to span across message block.

The basic structure of all DIA headers is presented below. As presented "<>" indicate mandatory elements, "()" indicate optional elements an "..." indicate repeat allowed. Further, the "FMX" designations refer to the message header types previously identified and "TTX denotes. (<Length> <Concatenation flag> <Type> (FM defined data)).

For TTX application-to-application message the structure is: (<FM0> (FM2) (FM8) (<FM64> (64text))... (Appl. Text)).

For TTX application-to-gateway application messages the structure is: (<FM0> (FM2) (FM4) (FM8) (<FM64> (64text))... (Appl. Text)).

For TTX message to TTX network management, the structure is: (<FM0> <<FM9> (9text)>...).

Finally, for internal TTX Switch to Switch messages, the header structure is: (<FM0> (Appl. Text)), where the FM0 function code is 2x or Cx.

Continuing, the general rules of implementation for DIA messages in the preferred embodiment are as follows. All internetwork messages are prefaced by a single FM0. Further, other header types can be optionally concatenated to the FM0. Also, headers should occur in ascending order by header type; i.e. FM0, FM2, FM4, FM8, FM9, FM64. Header and text length values are

carried as binary values. Numeric fields contained within DIA headers are carried with the most significant values in the left-most byte(s).

Further, long gateway messages (greater than 1K bytes including headers) are sliced up into blocks. This segmentation is indicated by the presence of the FM2 Header. In the preferred embodiment, the current block number of the FM2 must be correctly set because it acts as a sequence number and provides a means to guarantee message integrity. In this regard, the total number of blocks field must be set correctly when sending the last block of a logical message. Receiving programs can determine end-of-message by testing block number = total number blocks. If the sender cannot pre-determine the total number of blocks in a beginning or middle of message block, the sender must place binary zeros in the total number of blocks field.

Still further, in the preferred embodiment, FM9 architected text may not span message blocks and may not be longer than 255 bytes. Additionally, FM64 architected text may not span message blocks and may not be longer than 512 bytes long. Yet further, only a single instance of FM2 and/or FM4 can be present in a message block. And, messages using FM9 or FM64 headers must be less than 1K bytes, and these messages should not be segmented into blocks.

Continuing with the DIA implementation rules, FM0 and FM2 must be present in each block of a multi-block message when being transported within the network system. Normal application message flow consists of a request/response pair. In normal processing, reception system applications send requests to host applications. Host applications return responses to these requests. The Reception System application initiates this dialogue. Sending nodes are responsible for inserting the proper "source id" (SID) and "destination id" (DID) into the FM0. Additionally, the communications manager (CM) of the reception system further described hereafter, acts on behalf of reception system transaction programs. Messages destined to the CM should be considered systems messages (FM0 FUNCTION = Cn). Messages destined to subordinate transactions on reception system 400 should be considered applications message (FM0 Function=0n). Receiving nodes are responsible for swapping SID and DID contents when returning a response. Still further, intermediate nodes (with the exception of CICS switches and Gateways) need only be aware of FM0 and FM2 headers when routing messages to other destinations. CICS switches must be cognizant of all header layouts so that they can find the displacement to the transaction id which is contained within the first four bytes of application text. And server switch 205 provides a facility which allows responses to requests to be deliverable for at least a minimum period after the request was sent, e.g., one minute.

Finally, the preferred embodiment, CICS switches pass all DIA FM headers on to their subordinate applications. The applications are then responsible for returning the headers (with the SID/DID swap) back to the switch for responses. Both fixed length and variable length message headers are supported by the DIA. It must be noted that variable length headers are designed so that only the last field within the header is variable in length.

With regard to mode of conversation under utility sessions, the server switch 205 may engage in multiple sessions with an external CICS. Messages originating from network users may be routed through any of these sessions. Users are not forced to use the same utility session pipe for each message outbound to CICS. Pipes may be selected dynamically based on loading factors. In a switch-driven environment CICS transactions may typically be initiated by means of start commands from the switch. In this arrangement, CICS transactions will pass outbound data back to the switch through a queue.

In accordance with DIA, the potentially dynamic nature of conversation routing dictates that CICS transaction programs not be written in a conversational mode. Rather, the transaction programs are preferably either pseudo-conversational or non-conversational. In this regard it should be noted that conversational transactions send a message and wait for a reply, and non-conversational transactions send a message and expect no reply. In the case of pseudo-conversational transactions, a

message is sent, but no reply is expected. However, such messages are coded so as to be able to accept user input in various stages of completion, thus mimicking conversational transactions.

As will be appreciated by those skilled in the art, communications may arise within network 10 that do not require the standards applied to DIA messages. However, non-DIA messages are allowed in the DIA structure. Particularly, non_DIA messages are designated by setting the length portion of the header (i.e., the first byte) to binary zero.

Considering header layout, and with input first to FMO headers, it should be noted that the FMO header provides routing information to both intermediate and boundary switches. In addition the FMO contains control fields which allow the sending application (which may be a switch) to communicate information to the switch which "owns" the destination application. When an originating application wishes to converse with an application resident on the other side of an utility session it must initially pass an FMO header with a function code representing an "begin session" to its controlling switch. The begin session code requests the assistance of any intervening switches in the establishment of an application session between the requestor and the destination application specified in the DID.

When either application session partner wishes to terminate its conversation the session partner must pass an FMO header to its

switch, specifying either a function code representing an "end session", or "end session abnormal", or "request terminate". These function codes request the assistance of any intervening switches in the termination of the application session between the requestor and the destination application specified in the DID. In this arrangement an end session function code is unconditional and does not require an acknowledgment. An end session abnormal function code is unconditional and does not require an acknowledgment. And, a request terminate function code is conditional and requires a positive acknowledgement. The positive acknowledgement to a request terminate is an end session. The negative acknowledgement to a request terminate is a function code representing "status Message".

Further, "status/return" function codes "system up", "system down", "echo", "system message" are used by corresponding applications in different regions of network 10 to determine application availability and user session status. Function codes are also used to designate end-to-end user message classes of service. These classes of service refer to a delivery requirement classification and are distinguished from SNA COS. Network class of service allows applications to specify whether or not responses to requests can be delivered after the standard timeout of server 205 has occurred.

In accordance with the invention, the DIA headers are arranged in a predetermined form base on their function. More

Particularly, FMO headers, also known as Type "O" headers are required for every message within the network. Header Type 0 provides information necessary for routing and message correlation. Its fields include:

Header Length - Length of header data including length field. Header Type - Bit 0 is header concatenation flag.
Bits 1 - 7 indicate current header type.

Function Code - Contains message function. Data Mode - Indicates attributes of message data.
The "response expected" bit should be turned off if no response is expected, for instance, when sending the response to a request. Source Id - Identification of end-user sending current message. Logon Sequence Number - number which in conjunction with source id provides unique identification of source when source is reception system. Message Sequence Number - used to correlate requests and responses. Destination Id - Identification of message destination.
All messages are routed by destination id. When responses to messages are sent back to original source, the source id and destination ID fields must be swapped. Text Length - length of all remaining data in the message to the right of this fields. (Includes length of

concatenated headers if any are present).

The layout for the Type 0 header is as follows: Header Type 0 layout: Byte 0 Header Length (hexadecimal) Byte 1 Header Type

bit 0 if '0' no other headers present

if '1' concatenated header present

bits 1 - 7 000 0000 Byte 2 Function Code; i.e.

Application message

(Class of Service)

Status/Return Code message

Begin Session

End Session (normal)

End Session (error)

Clear Request (request terminate)

System Up

System Down

Echo

System Message

Prepare to bring System Down Byte 3 Data Mode (bit flags)

bits 0 - 7 .... Compaction

.... Encription

.... Response Expected

.... Response

.... Unsolicited Message

.... Logging required

.... Timeout Message Required

Source ID

bits 0 - 7 Region ID (hexadecimal)

bits 8 - 19 xxxx xxxx xxxx Unit: Source application id if in Application mode

xxxx xxxx xxxx Unit: Source Concentrator unit if in Reception System mode

bits 20 - 23 xxxx Id Mode e.g.,

Reception mode

Reception mode

Server 205 Application mode

Server 205 Application mode

Reserved

bits 24 - 31 xxxx xxxx Sub-unit ID (hexadecimal)

Byte 8 Logon Sequence Number (hexadecimal) Byte 9 Message Sequence Number (hexadecimal) Bytes 10 - 13 Destination ID

bits 0 - 7 Region ID (hexadecimal)

bits 8 - 19 xxxx xxxx xxxx Unit: Destination application ID if in Application mode

xxxx xxxx xxxx Unit: Destination Concentrator if in

		Reception System mode
bits 20 - 23	xxxx	Id Mode; e.g.,
		Reception mode
		Reception mode
		Server 205 Application mode
		Server 205 Application mode
		Cache 302 Application mode
		Reserved
bits 24 - 31	xxxx xxxx	Sub-unit ID (hexadecimal)
Bytes 14 - 15	Text Length	

With regard to FM2 or Type 2 messages, when an application is transmitting a large message, the sending application or its controlling switch can slice up the message into a number of smaller messages. The FM2 message header is used to indicate how these smaller messages can be reassembled into a single logical message by the receiving application or its controlling switch.

In preferred form, the maximum logical message size is 64K. The maximum message block size is 1K including all headers. Block sequence numbers in the FM2 range from 1 to a maximum of 255. And a single block message will be sequenced as block 1 of 1 in the FM2.

When network objects are large (greater than 1K bytes) they are sliced up into smaller blocks. Each object block is prefaced by an "object block header". Object block headers are found in the application text portion of a message. Object block headers provide sequencing information to cache/concentrator 302. The presence of an object block header does not obviate the requirement for an FM2 DIA header, except in the case of messages from the cache/concentrator down to RS 400. Both an object block header and a FM2 may be present in a message. Sequence numbering within object block headers ranges from 0 to 255. A single block Object will be sequenced as block 0 of 0.

Messages larger than 1K are subdivided into 1K blocks when being transmitted between the server switch 205, cache/concentrators 302, and reception systems 400.

Header Type 2 (FM2) message header contain information about this dividing of large messages and is useful when re-constructing large messages. The fields for an FM2 message header are as follows: Header Length - length of header data including length field. Header Type - Bit 0 is header concatenation flag.

Bits 1 - 7 indicate current header type. Number of Blocks - total number of blocks used to transmit the logical message. If the total number of blocks cannot be determined at the time the first or middle blocks of a message are being sent,

this field may be set to zero.

The last block of a message must contain the correct total number of blocks. Block

Number - number of the current message block being transmitted.

Further, the layout for a Type 2 header is as follows:

Byte 0 Header Length (hexadecimal) Byte 1 Header Type

bit 0 if '0' no other headers present

if '1' concatenated header present

bits 1 - 7 000 0010 Byte 2 Number of Blocks (hexadecimal) Byte 3 Current Block Number (hexadecimal)

With regard to FM4 type headers, also referred to as Type "4", these headers have been designed for communications between network gateway interface applications and external computer systems. For Type 4 Headers, the fields are as follows: Header Length - length of header data including length field. Header Type - Bit 0 is header concatenation flag.

Bits 1 - 7 indicate current header type. Network User - a seven byte field containing the internal ID ID of the network user on whose behalf a

conversation is being held with the external

computer system. External Data - Reserved Mode Correlation Id - a field reserved for use by the external

computer system. The contents of this field will initially be set to zero when a conversation is initiated across a gateway. The external system may then set the contents of this field to any value desired. Subsequent messages originating from TTX within he bounds of a virtual subscriber to external host session will echo the contents of the Correlation Id field back to the external system.

The layout for a Type 4 header is as follows:

Byte 0 Header Length (hexadecimal)

Byte 1 Header Type

bit 0 if '0' no other headers present

if '1' concatenated header present

bits 1 - 7 000 0100 Bytes 2 - 8 Network User Id (ASCII) Byte 9 External Data Mode

0000 0000 Reserved Bytes 10 - n Correlation Id (binary, max length=8 bytes)

Next are FM8 or Type 8 headers. Type 8 headers have been designed to provide secondary routing destinations. Their fields are as follows.

Header Length - length of header data including length field. Header Type - Bit 0 is header concatenation flag.

Bits 1 - 7 indicate current header type.

Secondary Destination - a symbolic name representing the ultimate

destination for the message.

The layout for Type 8 header is:

Byte 0	Header Length (hexadecimal)	
Byte 1	Header Type	
bit 0		if '0' no other headers present
		if '1' concatenated header present
		bits 1 - 7.......... 000 1000
Bytes 2 - 9	Symbolic Destination Name	

For FM9 or Type 9 headers, the header has been designed to communicate to a VTAM application which provides various network management support functions. More specifically, the VTAM application has been developed in order to provide: a general network management interface which both supports the network (by means of the DIA) and simplifies its maintenance. Additionally, VTAM application provides data transfer and remote command functions, the ability to write to, and read from, a centrally located and maintained database in order to archive statistics and other inter-network messages, and formatting of binary data into Hexadecimal Display.

In the case of Type 9 headers, the fields are:

Header Length - length of header data including length field. Header Type - Bit 0 is header concatenation flag.

Bits 1 - 7 indicate current header type.

Function Code - indicates general message type. Reason Code - indicates message content. Flags - indicates application action to be performed. Text Length - indicates length of subsequent text message.

(Not including possible concatenated headers)

The layout for type 9 headers is:

Byte 0 Header Length (hexadecimal)

Byte 1 Header Type

bit 0 if '0' no other headers present

if '1' concatenated header present

bits 1 - 7 000 1001 Byte 2 Function Code; e.g.

Command

Statistics

Alert

Control Byte 3 Reason Code

Backbone Alerts Message

Reception-originated Alerts Message Byte 4 Flags

bits 0 - 3 1... Store by Key - 8 char name follows

.1.. Retrieve by Key - 8 char name follows

..1. Data is Binary

...1 Data is (ASCII)

..00 Data is EBCDIC

bits 4 - 7 Reserved Byte 5 Text Length

(if Flags = 1... or .1.. then chars 0 - 7 should be the storage key. It is recommended that record storage keys initially be the same as the Resource Name to which the data pertains.

In the case of FM64 or Type 64 headers, the headers are used to transmit error and status messages between applications. Intermediate nodes need not examine the contents of the FM64 headers except in the case of the CICS switch which must obtain the displacement to the application text. If applications subordinate to an application subsystem are not available, the subsystem would strip the application text from the message, concatenate an FM64 message to any other headers which are present in the inbound message, and return the message to its original source.

Header Type 64 has been designed for the communication of status information between users, and prefaces architected message text. The fields for Type 9 headers are: Header Length - length of header data including length field.

Header Type - Bit 0 is header concatenation flag.

Bits 1 - 7 indicate current header type. Status

Type - indicates type of status communicated such as

status request or error. Data Mode - indicates whether message text is ASCII or

EBCDIC Text Length - Length of subsequent message text

(Not including possible concatenated headers).

The header Type 64 layout is:

Byte 0 Header Length (hexadecimal)

Byte 1 Header Type

bit 0 if '0' no other headers present

if '1' concatenated header present

bits 1 - 7 100 0000 Byte 2 Status Type

Information

Status Request

Error

Terminate Byte 3 Data Mode; e.g.

EBCDIC

ASCII

Binary Bytes 4 - 5 Text Length

In accordance with the invention, it has been determined that in some cases it is desirable to pre-define certain application level message formats so that they may be consistently used and interpreted. The following discussion is devoted to architected message text formats which are processed at the application level. For FM9 message text, in order to accommodate "Reliability/Serviceability/Availability" RAS functions within

network 10, a fixed format for "alerts" is defined in the preferred embodiment. Particularly if it is defined as message text following an FM9 header. The FM9 Function Code Alerts Message would be as follows:

Byte 0	Reserved value
Byte 1	System Origin
Byte 2	Internal/External flag
Byte 3-5	Message Originator
Byte 6 - 9	Message Number
Byte 10	Severity indicator; e.g.
	Error
	Information
	Severe Error
	Recovery Successful
	Warning Byte 11
Reserved value Byte 12 - 14	Error Threshold

For FM64 message text, the application message text is always prefaced by the appropriate header which indicates whether message text is ASCII or EBCDIC.

The FM64 message text fields are as follows:

Action Field - indicates type of operator or application action to be performed Module Name - Sending application Id

Format of this field is SSSTnnnn where

SSS = sender initials

T = type 0 = Network standard for all

gateways 1 = non-standard, gateway specific

nnnn = Sender Site number

Reference Number - Number assigned by sender for reference This number is used to indicate specific error codes if the message is an error message (FM64 stat type 8). This number is used to indicate specific commands if the message is a status request (FM64 stat type 4). Text - Alphanumeric (Printable) text

The FM64 Message Text layout is:

Byte 0 Action Field (alphanumeric), e.g.

Action

Decision

Information

Wait Bytes 1 - 8 Module Name (alphanumeric)

Bytes 9 - 12 Reference Number

(display numeric)

Default

request user status

user active

user inactive

user inactive - retry after interval

store in user mailbox

cache to server link failure

request appl status

Server to Host failure

appl active

appl inactive

appl inactive - retry after interval

message was undeliverable

response was timed out

Syncpoint

Checkpoint

Delay

appl. error codes Bytes 13 - n Text (alphanumeric)

Turning next to co called "Backbone States", as will be described below, application sessions may be used as pipes for user transaction traffic. In this regard, it is desirable to establish a set of protocols to be used between originating users and destination users. Further it is important for intermediate nodes to be aware of the status of connectivity with adjacent nodes and specifies some actions to take when messages are known to be undeliverable.

In this context, it is to be noted that the system up" message is used to signal the start of application traffic between the

switch applications. The originating application transmits an FM0 with a system up function code and response expected. The receiving application swaps the SID/DID, sets the Response bit on, and returns the message. If the receiving application is not available no response will be returned and the message will time out.

In the case of "system down" messages, the message is used to prepare the termination of the session between switch applications. The originating application transmits an FM0 with a session down function code and response expected. The originating application sends an FM64 with "status type=terminate",and data mode=EBCDIC. FM64 text follows the header with "action field"=A (Action), "module name"=SSSx0nnnn, "reference number"=0, Text=((timestamp=HHMMSS), Number of current users = NNNNN). The intended result is that the originating application will not accept any messages inbound to the utility session. The responding application will then have the opportunity to return outstanding responses across the utility session. The responding application then returns an FM0 with System Down back to the originating application.

For each "echo" messages, the echo message may be used to determine whether a major application is still available. Specifically, the originating application sends an application message to its gatewayed partner using a FM0 with an echo function. The destination application swaps the SID/DID, set the response bit

or[illegible]nd returns the message otherwise untouched, thus effecting echo.

For "APPL status request messages, the message is used to determine the status of a major application between nodes.

Continuing, for "unsolicited application status posting" messages these messages are used for transmission of application status messages by unsolicited application (No response expected) across a nodes. For the message, the originating application wishes to post an application status to its partner in another node. This message may be on the behalf of the originating application itself or on behalf of another application.

Turning next to user to internal APPL messages, and with regard to "session beginning", it is to be noted these messages normally arise at the start of conversation between a user and an internal application. For them the network user sends an FMO with a "begin session" function code and "response expected". The responding application swaps the SID/DID, supplies a "correlation Id", and returns both the FMO with the response bit set.

In the case of rejection of a conversation initiation requests, the originating application transmits an FMO with a "begin session" function code and "response expected". The responding application swaps the SID/DID, and returns the FMO with the response bit set as well as a function code of "abend" session.

For "applications" messages, these messages normally arise at the middle of conversation between a network user and an internal application. In this case, the originating user transmits and FMO with an "application" message function code, and "response expected". The responding application swaps the SID/DID, sets the response bit on and returns the response.

"End session" messages typically arise in connection with unconditional termination of user/internal application sessions. The originating transmits an FMO with an "end sessions" function code. Here however, no response is expected from the corresponding application.

For an "end session abnormal" message, the message unconditionally terminates an application conversation "abend"

Continuing, "request terminate" messages cause conditional termination of session with an internal application.

For messages concerning "rejection of a request due to link failure", in the case of server 205 to host link, the originating application transmits and FMO with "response expected". The message is intercepted by server 205 which recognizes it as undeliverable. A server 205 application returns the message with an FM64 message after stripping the application text.

For messages concerning rejection of request due to link failure, in the case of communication between the cache/concentrator 302 and server 205, the originating application transmits an FMO with Response Expected. The message is intercepted by the cache/concentrator 205 which recognizes it as undeliverable. A cache/concentrator application returns the message with an FM64 message after stripping the application text.

For messages concerning "conditional terminate rejected", the message is issued where a conditional termination of application conversation is not accepted by partner application.

For "user continuity posting" messages, the message is used where the originating application wishes to post the status of a user to its partner application across the gateway 210.

Continuing, for "user continuity requests", the message is used where an external application requests logon status of a particular network user.

In the case of "application error" messages, the messages is used where transmission of application error message by responding application is required.

Still further, for "timeout scenarios", and specifically, "timeout scenario with timeout response required", the

originating user sends an application message to an internal application with "data mode" = "response expected" and "timeout response" required. The originating switch sets a timer for each "response expected" outbound message. If a response is not received before the switch timeout value is reached the switch 205 sends a message with an FM64 header having a "timeout reference" code to the originating application.

For "response occurs after timeout" messages, the originating user sends an application message to an internal application with "response expected". The originating switch sets a timer for each "response expected" outbound message. If a response is received after the timeout value is exceeded, server 205 switch routes the message to a server 205 application which may log the message as non-deliverable, ship the message to the user, or drop it depending on the FMO class of service option specified on the original request message.

In the case of "maximum resources scenario" messages, the originating user transmits a message to a destined internal application. The destination switch determines that no resources are currently available to support the transmission, and returns the message to the originator, after inserting an FM64 with a "status=error and FM64 text with an "action=wait. The originating user may then retry or take other action.

Finally, the following graphic example illustrates normal message flow.

```
Use              Origin         Destination            APPL
                 Switch         Switch
------------------------------------------------------
  |                |                 |                 |
  |                | --------------->|                 |
  |                | System Up       |                 |
  |                | <-------------- |                 |
  |                | System Up RSP   |                 |
  |                |                 |                 |
  | ------------------------------------------------->|
  | Begin Session  (User 1)          |                 |
  | <------------------------------- |---------------  |
  |                |                 |Response User 1  |
  |                |                 |                 |
  | ------------------------------------------------->|
  | Begin Session  (User 2)          |                 |
  | <------------------------------- |---------------  |
  |                |                 |Response User 2  |
  |                |                 |                 |
  | ------------------------------------------------>  |
  |  Application Message User 1      |                 |
  | <----------------------------------------------->  |
  |                                  Response User 1   |
```

```
                        |                         |                      |
|----------------------------------------------------------------------|
|----------------------------------------------------------------------|
|                       |                         |                      |
|  ------------------------------------------------|----------------->|
|  End Session (User 1)                            |                      |
|                       |                         |                      |
|  ------------------------------------------------|----------------->|
|  Request Terminate (User 2)                      |                      |
|  <-----------------------------------------------------------------  |
|                       |                         |Response User 2    |
|                       |                         |                      |
|                       |  ---------------->|                      |
|                       |  System Down            |                      |
|                       |  <---------------       |                      |
|                       |  System Down RSP        |                      |
```

Turning next to messages passed over gateways 210, the normal exchange of messages between the network and external parties occurs between two applications; i.e., the server 205 network message handler (NMH). The server Switch 205 is an application which is written and maintained by network 10 and resides on it. The message handler resides on the other side of gateway 210 from network 10 and may be written and maintained by the external party; i.e., suppliers of information to network 10 such as Dow Jones.

The session between the two applications is used as a pipe for the communications between many network users and a variety of applications external to the network. In this design, the switch server 205 has three primary responsibilities. It must pass network originated messages across the gateway to the network message handler. It must distribute messages returning across gateway 210 to the appropriate network applications or users, i.e. RS 400. Additionally, it must manage the continuity of a network user session with the external service provider. Typically, users enter into a conversation with a set of transactions. This set of transactions is referred to as a task. These tasks are called user sessions. The boundaries of these tasks are indicated by begin session/end session flags.

The network message handler also has several responsibilities. It must pass externally originated messages across gateway 210 to the switch server 205 at network 10. It must distributed messages returning across gateway 210 to the appropriate external applications. And, it must be able to communicate the availability of external applications to network switch server 205.

With regard to gateway messages, in the case of "application to application" messages, and for "system up" messages, the system up message is used to signal the start of application traffic between switch 205 and the network message handler. The originating application transmits an FMO with function code "system up", and "response expected". The receiving application

swaps the SID/DID, sets the response bit on, and returns the message. If the receiving application is not available no response will be returned and the message will time out.

Continuing for gateway "system down" messages, the system down message is used to prepare the termination of the session between the switch 205 and the NMH. The originating application transmits an FMO with function code "session down" and "response expected. The originating application sends an FM64 with "status type"="terminate","data mode"="EBCDIC". FM64 Text follows the header with "action field"="A" (Action), "module name"="SSSx0nnnn", "reference number"="0", "text"=((timestamp=HHMMSS), number of current users = NNNNN). The intended result is that the originating application will not accept any messages inbound to the utility session. The responding application will then have the opportunity to return outstanding responses across the utility session. The responding application then returns an FMO with system down back to the originating application.

Further, for "prepare to bring system down" messages, the message is used to prepare the termination of the session between the Switch 205 and the NMH. The originating application transmits an FM0 with function code "prepare system down". The responding application transmits an FM0 with function code "session down" and "response expected". The responding application sends an FM64 with "status type"="terminate", "data

mode"="EBCDIC". FM64 Text follows the header with "action field"="A" (action), "module Name"="SSSx0nnnn", "reference number"="0", "text"=((Timestamp=HHMMSS), number of current users = NNNNN). The intended result is that the responding application will not accept any messages inbound to the utility session. The originating application will then have the opportunity to return outstanding responses across the utility session. The originating application then returns an FM0 with "system down" back to the responding application.

For "echo" messages, the message may be used to determine whether a major application is still available. The originating application sends an application message to its gatewayed partner using a FM0 with function echo. The destination application swaps the SID/DID, set the response bit on and returns the message otherwise untouched.

In the case of "APPL status request", the request is used to determine the status of a major application across the gateway.

Continuing, for "unsolicited application status posting messages, the message is used for transmission of application status messages by unsolicited applications no response expected across a gateway. In this case the originating application wishes to post an application status to its partner across the gateway. This message may be on the behalf of the originating application itself or on behalf of another application.

network to use "external APPL" messages, within the case of "begin session" messages, the message is used for normal start of conversation between a and an external application. The user, i.e. RS 400 sends an FM0 with function "begin session" and "response expected", as well as an FM4 with null value in the "correlation id". The responding application swaps the SID/DID, supplies a Correlation ID, and returns both the FM0 with the response bit set and the FM4.

For rejection of a conversation initiation request, the originating application resident application, transmits an FM0 with function Begin Session and Response Expected as well as an FM4 with NULL value in the Correlation ID. The responding application swaps the SID/DID, and returns the FM0 with the response bit set as well as a function code of ABEND session. The responding application also returns the FM4.

Further, for "applications" message, the message is used for normal middle of conversation between a network user and an external application. The originating user transmits an FM0 with function code "application" message, and "response expected". It also supplies the TTXUID and the correlation id received on the begin session response back to the corresponding application across the gateway. The responding application swaps the SID/DID, sets the response bit on and returns the FM0 and FM4.

"end session" message, the message is used for unconditional termination of user/external application sessions. The originating user transmits an FM0 with function code "end session", no "response expected". Additionally it sends an FM4 containing the TTXUID and the echoed "correlation id" in an FM4. No response is expected from the corresponding application.

For "end session abnormal" messages, the message is used for unconditional termination ABEND of gatewayed application conversation.

In the case of "request terminate", the message is used for conditional termination of user session with an external application.

For "conditional terminate rejected" messages, the message is used for a conditional termination of application conversation not accepted by partner application across a gateway.

For "user continuity posting" messages, the message is used where the originating application wishes to post the status of a user to its partner application across the gateway.

In the case of "user continuity" request, external application requests logon status of a particular user, i.e. RS 400.

For "application error" messages, the message is used for transmission of application an error message by responding application across a gateway.

In the case of "delayed response" messages, the originating application sends an application message to its gatewayed partner using the minimally a FM0 and a FM4 FM64 may be present. The destination switch signals an application on the originating side that the response may be slow by sending a FM0 with function code "status/return", the response bit is not set. The FM4 is returned, and an FM64 "status", FM64 text "Action"="Information" is also sent. Slow response may be due to a number of factors such as function shipping requirements or many I/Os. In parallel, the gateway partner application processes the message according to normal flow.

For "timeout scenario", the originating user sends an application message to an external application with "response expected". The switch server sets a timer for each "response expected" outbound message. If a response is received after the timeout value is exceeded, the TPF switch routes the message to a TPF application which may log the message as non-deliverable, ship the message to the user, or drop it depending on the FM0 class of service option specified on the original request message.

For the "maximum resources scenario" messages, the originating user transmits a message to a destined external application. The network message handler determines that no resources are currently available to support this transmission. The network message handler returns the message to the originator, after inserting an FM64 with a "Status"="Error" and FM64 text with an "action=wait". The originating user may then retry or take other action.

Finally, an example illustrates normal message flow.

```
________NETWORK_________          __________External____________  |
|        |                             | User             SERVER
205          TMH           APPL
-----------------------------------------------------
  |               |                 |                 |
  |               | --------------->|                 |
  |               | System Up       |                 |
  |               | <-------------- |                 |
  |               | System Up RSP   |                 |
  |               |                 |                 |
  | ---------------------------------------------------->|
  | Begin Session  (User 1)         |                 |
  | <------------------------------ |---------------  |
  |               |                 |Response User 1  |
  |               |                 |                 |
  | ---------------------------------------------------->|
  | Begin Session  (User 2)         |                 |
```

```
<-------------------------------- |---------------- |
|                |                 |Response User 2 |
|                |                 |                |
| -------------------------------------------------> |
|  Application Message User 1      |                |
| <------------------------------------------------> |
|                                  Response User 1  |
|                |                 |                |
|----------------------------------------------------|
|----------------------------------------------------|
|                |                 |                |
| --------------------------------|---------------->|
| End Session (User 1)             |                |
|                |                 |                |
| --------------------------------|---------------->|
| Request Terminate (User 2)       |                |
| <------------------------------------------------ |
|                |                 |Response User 2 |
|                |                 |                |
|                | ---------------->|                |
|                | System Down     |                |
|                | <---------------  |                |
|                | System Down RSP |                |
```


And, the following is an example that illustrates premature loss of user connectivity due to the loss of connection between

the network switch server 205 and a cache/concentrator 302. In this case, an application peripheral to switch 205 posts the user status inactive to the NMH using an FM64 Ref=0008 user inactive. External application reaction to this posting is implementation dependent. In this example, the external application returns outstanding responses using the FM64 "ref"="mailbox option".

```
________Network___________      __________External____________
|                          |        |
| Server Appl     Server Switch     TMH                APPL
-----------------------------------------------------------------
--
  |               |                 |                 |
  | ------------------------------->|                 |
  | User Continuity Posting         |                 |
  |               |                 | -------------->|
  |               |                 |                 |
  |               |                 |                 |
  | <----------------------------------------------- |
  |      Outstanding Appl RSP with Mailbox option     |
  |               |                 |                 |
  |               |                 |                 |
  |               |                 |                 |
  |               |                 |                 |
```

In accordance with the invention, in order to enable the manipulation of the network objects, the application programs

necessary to support the interactive text/graphic sessions are written in a high-level language referred to as "TBOL", TRINTEX Basic Object Language (TRINTEX is the company developing the technology that constitutes this invention). TBOL is specifically adapted for writing the application programs so that the programs may be compiled into a compact data stream that can be interpreted by the application software operating in the user personal computer, the application software being designed to establish the network Reception System 400 previously noted and described in more detail hereafter.

In accordance with the invention, the Reception System application software supports an interactive text/graphics sessions by managing objects. As explained above, objects specify the format and provide the content; i.e., the text and graphics, displayed on the user's screen so as to make up the pages that constitute the application. As also explained, pages are divided into separate areas called "partitions" by certain objects, while certain other objects describe windows which can be opened on the pages. Further, still other objects contain TBOL application programs which facilitate the data processing necessary to present the pages and their associated text and graphics.

As noted, the object architecture allows logical events to be specified in the object definitions. An example of a logical event is the completion of data entry on a screen; i.e., an

application page. Logical events are mapped to physical events such as the user pressing the <ENTER> key on the keyboard. Other logical events might be the initial display of a screen page or the completion of data entry in a field. Logical events specified in page and window object definitions can be associated with the call of TBOL program objects.

Reception Systems 400 is aware of the occurrence of all physical events during the interactive text/graphic sessions. When a physical event such as depression of the forward tab key corresponds to a logical event such as completion of data entry in a field, the appropriate TBOL program is executed if specified in the object definition. Accordingly, the TBOL programs can be thought of as routines which are given control to perform initialization and post-processing application logic associated with the fields, partitions and screens at the text/graphic sessions.

Reception System 400 run time environment uses the TBOL programs and their high-level commands called verbs to provide all the system services needed to support a text/graphic session, particularly, display management, user input, local and remote data access.

In accordance with the invention, the TBOL programs have a structure that includes three sections: a header section in which the program name is specified; a data section in which the data structure the program will use are defined; and a code section in

which the program logic is provided composed of one or more procedures. More specifically, the code section procedures are composed of procedure statements, each of which begins with a TBOL key word called a verb.

In accordance with the invention, the name of a procedure can also be used as the verb in a procedure statement exactly as if it were a TBOL key word verb. This feature enables a programer to extend the language vocabulary to include a customized application-oriented verb commands.

Continuing, TBOL programs have a program syntax that includes a series of "identifiers" which are the names and labels assigned to programs, procedures, and data structures.

An identifier may be up to 31 characters long; contain only uppercase or lowercase letters A through Z, digits 0 through 9, and/or the special character underscore (_); and must begin with a letter. Included among the system identifiers are: "header section identifiers" used in the header section for the program name; "data section identifiers" used in the data section for data structure names, field names and array names; and finally, "code section identifiers" used in the code section for identification of procedure names and statement labels.

The TBOL statement syntax adheres to the following conventions. Words in uppercase letters are key words and must be entered

exactly as shown in an actual statement. When operand are allowed, descriptive operand names and lowercase letters follow the key word. In this arrangement, operand names or literals are entered in an actual statement. Operand names enclosed in square brackets ([ ]) are optional and are not required in an actual statement. Operand names separated by a bar (|) mean that one, and only one, of the separated operand can be included in an actual statement. Operand names followed by an ellipsis (...) can be entered 1 or more times in an actual statement. Model statement words not separated by punctuation must be separated by at least one blank (or space character) in actual statements. Model statement punctuation such as comma (,), semicolon (;), less-than sign (<), equal sign (=), greater-than (>), and parentheses (()) must be included where shown in actual statements. Square brackets ([]), bars (|), and ellipses (...) should not be included in actual statements.

An example of a model statement would be as follows:

```
GOTO_DEPENDING_ON index,label (.label...);
```

This model says that a valid GOTO_DEPENDING_ON statement must begin with the word "GOTO_DEPENDING_ON" followed by at least one blank. Thereafter, an "index" and a "label" separated by a comma must be included. The index and at least one label are required. Additional labels may also be used, provided each is preceded by

a comma. Further, the statement must have a semicolon as the last character.

Comments can be included in a TBOL program on a statement line after the terminating semicolon character or on a separate comment line. Comment text is enclosed in braces ({}). For example: {comments are enclosed in braces}. Comments can be placed anywhere in the source code stream since, in accordance with the invention they are ignored by the TBOL compiler. Additionally, blanks (or space characters) are ignored in TBOL statement lines except where they function as field separators.

As noted, TBOL programs have a structure that includes a header section, data section and code section. More particularly, every TBOL program must have a header section. The header section contains a PROGRAM statement. The PROGRAM statement contains the key word PROGRAM followed by the name of the program. For example: PROGRAM program_name; where "program_name" is an identifier; i.e., the name of the program.

Accordingly, the header section for a TBOL program called LOGON would look like as follows:

```
PROGRAM LOGON:      {User logon program}
```

The data section in a TBOL program begins with the key word DATA which is followed by data structure statements. The

structure statements contain the data structure definitions used by the program. If the data structure does not have to be defined for the program it can be omitted. However, if a TBOL program does not include a data section, it must use a more restricted structure, more fully explained hereafter.

As an example, the data syntax would be: DATA structure [structure...] where "structure" is a data structure statement. The data structure statement contains a definition, which consists of the data structure name followed by an equal sign and then the names of one or more variables. For example: structure_name=variable_name [,variable_name...]; where "structure_name" is an identifier; i.e., the name of the data structure; and "variable_name" is an identifier for the variable; i.e., the name of a variable.

All of the variables in the data structures are defined as string (or character) variables. TBOL string variables are of two kinds, fields and arrays. In the case of filed definitions, a variable field is defined with and identifier; i.e., the name of the field. No data type of length specification is required. An individual field is referenced by using the field name. Further, subsequent fields can be referenced by using a field name followed by a numeric subscript enclosed in parentheses (()). The subscript however, must be an integer number.

A field name followed by a subscript refers to a following field in the data section of a TBOL program. The subscript base is 1. For example, if a field CUST_NBR were defined, then CUST_NBR refers to the field CUST_NBR, CUST_NBR(1) also refers to the field CUST_NBR and CUST_NBR(2) refers to the field following CUST_NBR.

In the case of array definitions, the TBOL array is a one-dimensional table (or list) of variable fields, which can be referenced with a single name. Each field in the array is called an element.

An array can be defined with an identifier, particularly, the name of the array, followed by the array's dimension enclosed in parentheses (()). The dimension specifies the number of elements in the array. By way of illustration, if an array is defined with a dimension of 12, it will have 12 elements. An individual element in an array is referenced by using the array name followed by a numeric subscript enclosed in parentheses (()). The subscript indicates the position of the element in the array. The first element in an array is referenced with a subscript of 1. The subscript can be specified as either an integer number or an integer register as described, hereafter.

With regards to variable data, data contained in variables is always left-adjusted. Arithmetic operations can be formed on character strings in variables if they are numbers. A number is

a character string that may contain only numeric characters 0 through 9, an optional decimal point, an optional minus sign in the left-most position, commas and the dollar sign ($).

When you perform an arithmetic operation on a character string, leading and trailing zeros are trimmed and fractions are truncated after 13 decimal places. Integer results do not contain a decimal point. Negative results contain a minus sign (-) in the left-most position.

Each field and each array element has a length attribute which is initialized to zero by the Reception System at program startup. The LENGTH verb, to be described more fully hereafter, can be used to set the current length of a field or array element during program execution. The maximum length of a field or an array element is 65,535.

Further, the maximum number of variables that can be defined in the data section of a TBOL program is 222. This number includes fields and array elements.

The following example data section contains five data structure statements, each defining a data structure. Each structure statement begins with the name of the data structure followed by an equal sign.

t, are the names of the variables which make up the structure. The variable names are separated by commas. The last variable name in each structure statement is followed by a semicolon which terminates the statement.

The third data structure given, i.e. SALES_TABLE, contains two arrays. The others contain fields. The last structure statement, i.e. WK_AREA is and example of a single line.

```
DATA                          {Key word DATA begins data section}
BILL_ADDR=                {data structure BILL_ADDR}
BILL_NAME,                         {field1 BILL_NAME}
BILL_ADDR1,                        {field2 BILL_ADDR1}
BILL_ADDR2,                        {field3 BILL_ADDR2}
BILL_ADDR3;                        {field 4 BILL_ADDR3}
SHIP_ADDR=                {data structure SHIP_ADDR})
SHIP_NAME,                         {field1 SHIP_NAME}
SHIP_ADDR1,                        {field2 SHIP_ADDR1}
SHIP-ADDR2,                        {field3 SHIP_ADDR1} SHIP_ADDR3,
{field4 SHIP_ADDR1}
SALES TABLE=                  (data structure SALES_TABLE)
MONTH QUOTA(12),                   {array1 MONTH_QUOTA}
MONTH SALES(12);                   (array2 MONTH_SALES)
MISC_DATA=                {data structure MISC_DATA}
SALESPERS_NAME,           {field1 SALESPERS_NAME}
CUST_TELNBR;                           {field2 CUST_TELNBR}
WK_AREA=TEMP1,TEMP1           {data structure WK_AREA}
```

Continuing, TBOL contains a number of predefined data structures which can be used in a TBOL program even though they are not defined in the program's data section. There are two kinds of TBOL-defined data structures, these are "system registers" and "external data structures".

In the case of systems registers, tree different types exist. The first type are termed "integer registers",and are used primarily for integer arithmetic. However, these registers are also useful for field or array subscripts. The second type are termed "decimal registers", and are used for decimal arithmetic. The third type are called, "parameter registers" and are used to pass the data contained in procedure statement operand when the name of a procedure is used as the verb in the statement rather than a TBOL keyword.

The variables defined in the data section of a program are string (or character) variables, and the data in them is kept in string format. In most cases there is no need to convert this data to another format, since TBOL allows substantially any kind of operation (including arithmetic) on the data in string form. As will be appreciated by those skilled in the act, this eliminates the clerical chore of keeping track of data types and data conversion.

There are some cases where it is desireable to maintain numeric data in binary integer or internal decimal format. For example,

an application involving a great deal of computation will execute more efficiently if the arithmetic is done in binary integer or internal decimal format data rather than string data. In these cases, data conversion can be performed by simple moving the numeric data to the appropriate register. When data is moved from a register to a variable, it is converted to string format.

Integer registers are special-purpose fields for storing and operating on integer numeric data in binary format. The integer registers are named I1 through I8. Numeric data moved to an integer register is converted to an integer number in binary format. Further, an attempt to move non-numeric data to an integer register will cause an error. The largest negative number an integer register can hold is -32,7767, while the largest positive number than can be held is 32,767. An noted arithmetic operations in integer registers will execute more efficiently than arithmetic operations in string variables.

Decimal registers are special-purpose fields for storing and operating on numeric data in internal decimal format. The decimal registers are named D1 through D8. Numeric data moved to a decimal register is converted to a decimal number in internal decimal format. An attempt to move non-numeric data to a decimal register will cause an error. The largest negative number a decimal register can hold is -9999999999999.9999999999999, while the largest positive number a decimal register can hold is 9999999999999.9999999999999. Additionally, decimal registers can

not be used as field or array subscripts. And, again, arithmetic operations in decimal registers will perform better than arithmetic operations in string variables.

As pointed out above, the code section of a TBOL program contains the program logic, which itself is composed of one or more procedures. In the logic, the procedures are expressed as procedure statements. Each procedure statement begins with a TBOL keyword called a verb which is followed by operand, or parameters containing the data on which the verb is to operate. The name of a procedure can be used as the verb in a procedure statement exactly as if it were a TBOL keyword verb. As noted this enables the creator of a TBOL program; i.e. the party creating the text/graphic session, to extend the language vocabulary to include his own application-oriented verb commands.

When a procedure is used as the verb in a procedure statement, TBOL saves the current parameter register values, and the parameter data in the verb operands is moved into the parameter registers where it is available to the "called" procedure. When the "called" procedure returns, TBOL restores the saved parameter register values.

Parameter registers are special-purpose fields for passing parameter data to "called" procedures. The parameter registers are named P0 through P8. When a procedure is "called" by using its name as the verb in a procedure statement, the current

Array definition segments 548 define the names and relative locations of fields in a row that makes up an array or table. The first row of fields must have been defined using field definition segments 516. The array definition provides a short-hand for specifying the replication of selected fields from the initial page. The structure of the array definition segment 548 is as follows:

<st> <sl> <#occurrences> <vertical gap> <field name> ...

where "#occurrences" is a one-byte field describing the number of rows to be generated to create the array (the first row is assumed to be generated from field definition segments 516); "vertical gap" is a NAPLPS point set operand (relative, invisible) containing the DY of inter-row spacing; and "field name" is a one-byte name (from the field definition) of the fields in a row of the array. **CUSTOM GRAPHICS SEGMENT**

Custom graphics segment 550 provides a means to package graphics commands. These graphics commands may be related to a field and initiated based on run-time conditions. The structure of custom graphics segment 550 is as follows:

<st> <sl> <id> <size> <NAPLPS>

where "id" is a one-byte identifier for this custom graphic; "size" is a three-byte NAPLPS operand specifying upper right

contents of P0 through P8 are saved. Further, data from the first operand in the procedure statement is placed in P1; data from the second operand is placed in P2; and so on, up to eight operands. If no operand, or less than eight operand are specified, the parameter registers corresponding to the missing operand are set to null. In accordance with this arrangement, the number of operand is placed in P0, and the "called" procedure is given control.

When control returns to the "calling" procedure from the "called" procedure, the previous contents of P0 through P8 are restored. Following execution of the "called" procedure, execution of the "calling" procedure continues.

The "calling" procedure can pass along its own parameters to the "called" procedure by naming parameter registers as operand. The TBOL internal stack can be used to pass additional data to the "called" procedure, or to pass data back to the "calling" procedure.

There are two kind of TBOL-defined external data structures; they are partition structures and global structures. With regard to partition external data structures, as noted above the screens displayed during a test/graphic session are called pages. As also noted, pages may be divided into separate areas called "partitions". Each page partition has its own predefined partition external data structure. Each partition external data

structure can contain up to 256 variables for data pertaining to that partition. A TBOL program associated with a particular partition has access to the partition's external data structure and the variables it contains. However, the program cannot access another partition's external data structure.

The variable in a partition external data structure are character string variables like those defined in the data section of a program. The variables within each partition external data structure are named &1 through &256. The DEFINE compiler directive enables the program to use meaningful names for these variables in the program source code.

Partition external variables are used to hold screen field data, program flow data and applications data. In the case of screen field data, when page and window objects are defined, the fields in the screen partitions are assigned to partition external variables. The TBOL Object Linker resolves these references and at program execution time the Reception System transfers data between the screen fields and their associated partition external variables. The TBOL program has access to the variables, which contain the data entered in the screen fields by the user, and the user has access to the screen fields of which contain the data placed in the variables by the program.

For program flow data, partition external variables are used to hold the object identifiers needed by a TBOL program for

transferring control. These may be page object identifiers for transfer to another text/graphic screen page, or window object identifiers needed to open a window on the current page. As in the case of screen field data, flow data values are placed in partition external variable by the TBOL Object Linker.

Finally, for application data, partition external variables can be used to hold partition-specific application data such as tables of information needed by the program to process the expected screen field input.

With regard to the global external data structure, the predefined global external data structure can contain up to 32,000 variables for TBOL system data. All TBOL programs have access to the global external data structure and the variables it contains. The variables in a global external data structure are character string variables like the ones one defines in the data section of a program. The global external variables are named #1 through #32,000. These variables are assigned and controlled by the TBOL database administrator which maintains a file of DEFINE compiler directive statements which assign meaningful names to the global external variables in use. In the preferred embodiment, the MS-DOS file specification for this file can, for example be TBOLLIB\TBOL.SYS. In this regard, the COPY compiler directive is used to copy TBOL.SYS into a source code input stream. Subsequent statements in the program source code can

reference the global external system variables by using the meaningful names assigned by the DEFINE statements in this file.

Examples of global external variables are: SUS_RETURN_CODE, which is assigned a return code value after the execution of certain TBOL program verb statements; SYS_DATE, which contains the current system date; and SYS_TIME, which contains the current system time.

With regard to the TBOL program code section, as noted above, every TBOL program must have a code section. The code section contains the program logic which is composed of one or more procedures. In accordance with this arrangement, a procedure begins with the keyword PROC followed by an equal sign (=) and then the name of the procedure. The body of the procedure is composed of procedure statements, ending with the END_PROC statement. For example:

```
PROC=proc_name statement [statement...] END_PROC;
```

where "proc_name" is an identifier; i.e. the name of the procedure, and "statement" is a TBOL procedure statement as described below.

In accordance with the invention, at program execution time, control is given to the first procedure in the program. This is the mainline procedure. From then on, the flow of procedure

execution is controlled by the logic contained in the procedures themselves.

Each procedure statement begins with a TBOL keyword called a verb. However, as noted above, the name of a procedure can also act as the verb in a procedure statement, exactly as if it were a TBOL verb. In such case, the data in any statement operand is moved into parameter registers and control is passed to the other procedure. No special linkage or parameter passing conventions are needed. As will be appreciated by those skilled in the art, this is a powerful feature which enables the application programmer to extend the language vocabulary to include his own library of application-oriented verb commands and commonly used procedures.

When control is transferred to another procedure, as noted, the "called" procedure returns control to the "calling" procedure with a RETURN or END_PROC statement, where RETURN and END_PROC are TBOL verbs described more fully hereafter. Upon return, the "calling" procedure's parameter data, if any, is restored in the parameter registers, and program execution resumes with the next statement. Recursive logic is possible by using the name of the current procedure as the verb in a procedure statement, thus causing the procedure to "call" itself.

In accordance with the design of TBOL, any procedure statement may be preceded with one or more identifying labels. A

label consists of an Identifier followed by a colon (:). For example:

(stmt_label:...) statement

where "stmt_label" is an Identifier, for the statement, and "statement" is a TBOL procedure statement.

Procedure statement labels are used for transferring control to another statement within the same procedure using a GOTO or GOTO_DEPENDING_ON statement (TBOL verbs described more fully hereafter).

GOTO and or GOTO_DEPENDING_ON statement can also be used to transfer control to another procedure. Transfer to another procedure is done by using the target procedure name as the verb in a statement.

Also in accordance with the design of TBOL, all procedural logic is constructed from statements designed to execute in three basic patterns: sequential, conditional, or repetitive. In the case of a sequential pattern, the sequential program logic consists of one or more procedure statements. In the case of a conditional pattern, the conditional program logic is constructed using IF...THEN...ELSE and GOTO_DEPENDING_ON key words, described more fully hereafter. Finally, in the case of a repetitive pattern, the repetitive program logic is constructed using

WHILE...THEN key words or IF...THEN...ELSE and GOTO key words also described more fully hereafter.

In accordance with the TBOL design, a procedure statement may contain operand following the verb. In the case of procedure statements, there are five types of procedure statement operand; data names; group data names; system registers, label identifiers, and literals. In this arrangement, data names are the names of variables, and data name operand can be either field names; field numbers with subscripts or array names with subscripts. In the case of filed names, a field name is the identifier used as the name of a variable in a data structure in the data section of the program, or the name of TBOL-defined variable in an external data structure.

For field names with subscripts, a field name followed by a subscript enclosed in parentheses (()) refers to a following field. The subscript must be an integer number expressed as a literal or contained in a variable field. The subscript base is 1. For example: CUST_NAME(1) refers to the field CUST_NAME, and CUST_NAME(2) refers to the field following CUST_NAME.

For array names with subscripts, an array name is the identifier used as the name of an array in a data structure in the data section of the program. An array name followed by a subscript enclosed in parentheses (()), refers to an individual element in the array. The subscript must be an integer number

expressed as a literal or contained in a variable field. The subscript base in 1, so the first element in an array is referenced with a subscript of 1.

In the case of procedure statement group data name operand, the group data names are the names of data structures or arrays. Group data names are used in statements where the verb allows data structures or arrays to be treated as a single unit. For example, the TBOL MOVE verb allows the use of group data name operand. If the names of two arrays as group data operand are used, the contents of each element in the source array is moved to the corresponding element in the destination array. Here the array names are specified without subscripts. However, if the names of two data structures as group data operand are used, the contents of each variable in the source data structure is moved to the corresponding variable in the destination data structure.

With regard to system register operands, they can be either integer registers I1 through I8, or decimal registers D1 through D8, or parameter registers P1 through P8.

In the case of label identifiers, the label identifiers are the identifiers used as procedure statement labels described above.

Continuing, literal operand can be either, integer numbers, decimal numbers or character strings. Where the literal operand

are integer numbers, the integer is composed of the digits 0 through 9. Where a negative integer is to be represented, a minus sign (-) is allowed in the left-most position. However, a decimal point is not allowed. Accordingly, the minimum value that can be represented is -32,767, and the maximum value is 32,767. Where the literal operand is a decimal number, the decimal number is composed of the digits 0 through 9 with a decimal point (.) where desired. A minus sign (-) is allowed in the left-most position. Thus the minimum allowable value is -9999999999999.9999999999999, and the maximum value is 9999999999999.9999999999999.

Further, where the literal operand is a character string, the character string is composed of any printable characters or control characters. Character strings are enclosed in single quotes ('). To include a single quote character in a character string, it must be preceded with the backslash character (\). For example: \'. To include a new-line character in a character string, the control character \n is used. For example; 'this causes a new line: \n'. To include binary data in a character string, the hex representation of the binary data is preceded with the backslash character (\). For example; 'this is binary 01110111:\77'.

The syntax of a complete TBOL program is illustrated in the following example program.

HEADER SECTION PROGRAM program_name;

```
DATA SECTION            DATA
      :                 data_structure_name-1=    {1st data structure}
      :                 .
      :                 variable_name_1,
      :                 .
      :                 variable names
      :                 .
      :                 variable_name_n;
      :                 .
      :                 data structures
      :                 .
      :                 data_structure_name_n=    {nth data structure}
      :                 .
      :                 variable_name_1,
      :                 .
      :                 variable names followed by commas
      :                 .
      :                 variable_name_n;

CODE SECTION            PROC proc_name_1=         {mainline procedure}
      :                 .
      :                 procedure statements
      :                 .
      :                 IF x = x THEN EXIT:       {if done,return to
      :                                           Reception System}
      :                 procedure statements
      :                 .
```

```
            END_PROC;                  {end of mainline
                                       procedure}
:           .
:           procedures
:           .
:           PROC proc_name_n=          {nth procedures}
:           .
:           procedure statements
:           .
:           IF x = x THEN RETURN;      {if done,return to
:           .                          "calling"procedure}
:           procedure statements
:           END-PROC;                  {end of nth
                                       procedure}
                                            {end of
                                            program}
```

In accordance with the invention, the TBOL compiler enables portability of TBOL programs. Specifically, the TBOL compiler is capable of generating compact data streams from the TBOL source code that can be interpreted by any reception system configured in accordance with the invention, i.e., a personal computer running the reception system application software. For this arrangement, the compiler input file containing the TBOL source code may have any name. For example, the extension .SRC can be used.

During the compilation, three files are generated. Their names are the same as the source code file; their extensions identify their contents. For example, when the file names INPUT.SRC is compiled the following files are generated by the compiler: INPUT.SYM which contains a symTBOL directory; INPUT.COD which contains the compiled code; and INPUT.LST which contains the listing.

In order to resolve an undefined procedure, the TBOL compiler automatically search the local MS-DOS directory TBOLLIB for a file named procname.LIB, where procname is the name of the unresolved procedure. IF procname.LIB is found, the compiler will automatically copy it into the source code stream after the program source text has ended.

In addition to the undefined procedures facility above noted, the TBOL compiler also may be caused to substitute one text string for another. This accomplished by a DEFINE directive.

Wherever the text pattern specified in operand 1 is found in the source code stream, it is replaced by the compiler with the text pattern specified in operand 2. The syntax for the procedure is:

```
DEFINE source_pattern,replacement_pattern;
```

where "source_pattern" is the text in the source code which the compiler is to replace, and "replacement_pattern" is the text the compiler will use to replace source_pattern.

If source_pattern or replacement_pattern contain any blank (space) characters, the text must be enclosed in single quotes (').

Further, the compiler can be made to eliminate certain text from the input source stream by using a null text string for the replacement_pattern ('').

It is to be noted that while DEFINE directives are normally placed in the data section, they can also be placed anywhere in the source code stream. For example, if the symTBOLic name CUST_NUMBER has been used in a TBOL application program to refer to a partition external variable named &6. The DEFINE statement DEFINE CUST_NUMBER,.&6 would cause the compiler to substitute &6 whenever it encounters CUST_NUMBER in subsequent statements.

As a further illustration , if the words MAX and MIN are defined with numeric values, DEFINE MAX,1279; and DEFINE MIN,500; MAX and MIN can be used throughout the program source code rather than the actual numeric values. If the values of MAX and MIN change in the future, only the DEFINE statements will need to be changed.

Still further, the compiler can also be caused to copy source code from some other file into the compiler input source code stream. This can be accomplished with a directive entitled COPY. With the use of the COPY directive, the source code contained in the file specified in operand 1 is copied into the source code stream at the point where the COPY statement is read by the compiler. For example, the syntax would be: COPY 'file_name'; where "file_name" is the name of the file containing source code to be inserted in the source code steam at the point of the COPY statement. In this arrangement, file_name must be enclosed in single quotes ('), and file_name must conform to the operating system file naming rules (in the current preferred embodiment, those of MS-DOS). Further, the file referenced in a COPY statement must reside in the TBOLLIB directory on the compilation machine. In accordance with the invention the COPY statement can be placed anywhere in the source code stream.

By way of illustration, the COPY statement COPY 'TBOL.SYS'; causes the compiler to insert source text from the file TBOL.SYS . This file is maintained by the TBOL Database Administrator, and contains DEFINE statements which assign meaningful names to the TBOL system variables in the global external data structure.

As shown in Fig. ___, 28 verbs are associated with data processing; 20 with program-flow; 7 with communications; 6 with file management, 6 with screen management; 3 with object management and 2 with program structure for a total of 72. Following

is alphabetical listing of the TBOL verbs, together with a description of its function and illustration of its syntax.

ADD

The ADD verb adds two numbers. Specifically, the number in operand 1 is added to the number in operand 2. Thus, the number in operand 1 is unchanged, while the number in operand 2 is replaced by the sum of the two numbers. The syntax for ADD is:

```
ADD number1,number2;,
```

where number1 contains the number to be added to number2. In this arrangement, number1 can be a data name; system register or literal number. As is apparent, number2 contains the second number, and is overlaid with the resulting sum. Number2 can be a data name or system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the add operation. In accordance with this embodiment, fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Negative results contain a minus sign (-) in the left-most position.

AND

The AND verb performs a logical AND function on the bits of two data fields. The logical product (AND) of the bits of operand 1 and operand 2 is placed in operand 2. Moving from left to right, the AND is applied to the corresponding bits of each field, bit by bit, ending with the last bit of the shorter field. If the corresponding bits are 1 and 1, then the result bit is 1. If the corresponding bits are 1 and 0, or 0 and 1, or 0 and 0, then the result bit is 0. In this arrangement, the data in operand 1 is left unchanged, and the data in operand 2 is replaced by the result.

The AND syntax is:

```
AND field1,field2;
```

where "field1" contains the first data field, which can be a data name, system register, I1 - I8 or P1 - P8 only, or a literal. Continuing, "field2" contains the second data fields, and the contents of field2 are overlaid by the result of the AND operation. Field2 can be a data name, a system register: I1 - I8 or P1 - P8 only.

As will be appreciated, the AND verb can be used to set a bit to 0.

CLEAR

The CLEAR verb sets one or more variables to null. The CLEAR statement may have either one or two operand. If only one operand is specified, it may contain the name of a field, an array or a data structure. If the operand contains a field name, then that field is set to null. If the operand contains an array name, then all elements of the array are set to null. If the operand contains the name of a data structure, then all fields and array elements in the data structure are set to null. If two operand are specified, then each operand must contain the name of a field. In this case, all fields, beginning with the field in operand1 and ending with the field in operand2, are set to null.

The syntax for CLEAR is CLEAR name1 [,name2]; where "name1" contains the name of a field, array, or data structure to be set to null. If "name2" is specified, name1 must contain a field name. Name1 can be a data name, group data name, or system register P1 - P8 only. Further, name2 contains the last field name of a range of fields to be set to null, and can be a data name, group data name, or system register P1 - P8 only.

CLOSE

The CLOSE verb is used to close a reception system file after file processing has been completed. By using CLOSE, the file named in operand 1 is closed. If no operand is specified, then all open files are closed. The CLOSE syntax is:

```
CLOSE [filename];
```

where "filename" contains the name of the reception system file to be closed. The file name "PRINTER" specifies the system printer. Otherwise, the name of the file must be a valid MS-DOS file specification; e.g., [drive:][\path\]name[.extension] File name can be a data name, or system register P1 - P8 only. When file processing is complete, the file must be closed.

CLOSE_WINDOW

The CLOSE_WINDOW verb is used to close the open window on the base screen and, optionally, open a new window by appending the partial operator _OPEN to the middle of the verb (as shown below). Specifically, by using CLOSE_WINDOW, open window on the base screen is closed. If no operand is specified, program execution continues with the next statement in the program which last performed an OPEN_WINDOW. If operand 1 is specified, the window whose object ID is contained in operand1 is opened, and program execution continues with the first statement of the program associated with the newly opened window object.

The CLOSE_WINDOW syntax is:

```
CLOSE_WINDOW [window-id];
```

where, "window-id" contains the object ID of a new window to be opened after closing the currently open window. A window-id can be a data name, system register P1 - P8 only, or a literal.

Similarly, the CLOSE_OPEN_WINDOW syntax, which causes a window to open upon the closing of the previous window, is:

CLOSE_OPEN_WINDOW [window - id]

The CLOSE_WINDOW verb can only be performed by a window program; i.e., a program associated with a window object. CLOSE_WINDOW is the method by which a window program relinquishes control. A window program can also close itself by performing one of the following verbs: NAVIGATE, TRIGGER_FUNCTION. Although a window program cannot perform a OPEN_WINDOW operation, it can use CLOSE_WINDOW to close itself and open another window. This process can continue through several windows. Finally, when a window program performs a CLOSE_WINDOW without opening a new window, program control does not work its way back through all the window programs. Instead, control returns to the non-window program which opened the first window. Program execution continues in that program with the statement following the OPEN_WINDOW statement.

CONNECT

The CONNECT verb dials a telephone number. The telephone number contained in operand 1 is dialed. The telephone line status is returned in the system variable SYS_CONNECT_STATUS.

The syntax for CONNECT is:

CONNECT phone_number;

where "phone_number" contains the telephone number to be dialed. Phone_number can be a data name, system register P1 - P8 only, or a literal.

DEFINE_FIELD

The DEFINE_FIELD verb is used to define a screen field at program execution time. From five to seven operand specify a single-line or multiple-line field within the currently active screen partition; i.e. the partition associated with the running program. The field is dynamically defined on the current screen partition.

The syntax for DEFINE_FIELD is:

```
        DEFINE_FIELD  name,row,column,width,height
[,object_id [,state] ];
```

where "name" is the field to receive the name of a partition external variable. When this statement is performed, a screen field is defined and it is assigned to a partition external variable. The partition external variable name is placed in the name operand. Name may be a data name, or system register P1 - P8 only.

Continuing "row" in the DEFINE_FIELD syntax contains the row number where the field starts. The top row on the screen is row number 1. Row can be a data name, system register P1 - P8, or a literal. "Column" contains the column number where the field starts. The leftmost column on the screen is column number 1. Column can be a data name, system register P1 - P8 only, or a literal. In the DEFINE_FIELD syntax, "width" contains a number specifying how many characters each line the field will hold. Width can be a data name, system register P1 - P8 only, or a literal. Further, "height" contains a number specifying how many lines the field will have. For multiple-line fields, each field line will begin in the column number specified in the column operand. Height can be a data name, system register P1 - P8 only, or a literal.

Yet further, in the DEFINE_FIELD syntax, "object_id" contains the object ID of a field post processor program that is to be associated with this field. Object_id can be a data name, system register P1 - P8 only, or a literal. Finally, for the DEFINE_FIELD syntax "state" contains a character string which is to be placed in parameter register P1 when the program specified in the object_id operand is given control. State can be a data name, system register P1 - P8 only, or a literal.

In the case of the DEFINE_FIELD verb, if the object-id operand is specified, then the post processor program object is obtained only on a "commit" event; avoiding the need for a

synchronous FETCH. Since DEFINE_FIELD defines a field only in the screen partition associated with the running program, a program can not define a field in some other screen partition with which it is not associated. Additionally, page-level processor programs which are not associated with a particular screen partition can not use this verb.

DELETE

DELETE is used to delete a reception system file for file processing. the file named in operand 1 is deleted. The syntax for DELETE is:

```
DELETE [filename];
```

where "filename" contains the name of the reception system file to be deleted. Filename can be a data name or system register P1 - P8. Filename must be a valid operating specification.

DISCONNECT

The DISCONNECT verb "hangs up the telephone", thus, terminating the telephone connection. The syntax for DISCONNECT is simply:

```
DISCONNECT; .
```

DIVIDE

The DIVIDE verb divides one number by another. The number in operand 2 is divided by the number in operand 1. The number in operand 1 is unchanged, however, the number in operand 2 is replaced by the quotient. If operand 3 is specified, the remainder is placed in operand 3. The syntax for DIVIDE is:

```
DIVIDE number1,number2  [,remainder];
```

where "number1" contains the divisor, i.e. the number to be divided into number2. Number1 can be a data name, system register, or literal number. Continuing, "number2" contains the dividend; i.e., the number to be divided by number1. The contents of number2 are overlaid by the resulting quotient. Number2 can be a data name, or a system register. And, "remainder" is a variable or system register designated to hold the remainder of the divide operation. Remainder can be a data name, or a system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the divide operation. Fractions will be truncated after 13 decimal places, while whole number will not contain a decimal point. Negative results will contain a minus sign (-) in the left-most position.

DO...END

The keyword DO specifies the beginning of a block of statements; the keyword END specifies the end of the block. A block of statements, bracketed by DO and END can be used as a clause in an IF or WHILE statement. In an IF statement, either the THEN clause or an optional ELSE clause can be executed, based upon the evaluation of a boolean expression. In a WHILE statement, the THEN clause is executed repetitively until a boolean expression is false.

The syntax for DO...END is:

DO...block...END;

where "Block" is any number of TBOL statements. As shown, the keyword DO is not followed by a semicolon, and the END statement requires a terminating semicolon.

EDIT

The EDIT verb gathers and edits data from multiple sources, then joins it together and places it in the specified destination field. Data from one to six sources, beginning with operand 3, is edited in accordance with the mask contained inperand 2. The edited data, joined together as a single character string is places in the output destination field specified in operand 1.

The EDIT syntax is EDIT output,mask,source [,source...];, where "output" contains the name of the destination field for the

edited data. After performance of the EDIT statement, the destination field will contain "sub-fields" of data; one for each source operand. Output can be a data name, or a register P1 - P8 only.

Continuing, "mask" contains a character string consisting of one edit specification for each source operand. Edit specifications are in the form: %[-][min.max]x, where "%" indicates the beginning of an edit specification; "-" indicates left-adjustment of the source data in the destination sub-field, and "min.max" are two numbers, separated by a decimal point, which specify the minimum and maximum width of the edited data in the destination sub-field, and "x" is an alpha character which controls the retrieval of data from the corresponding source operand. Further, "x" can be a "d" to indicate a digit, characters retrieved from the corresponding source operand are converted to integer format; or "x" can be an "f" to indicate floating point, characters retrieved from the corresponding source operand are converted to a decimal format; or an "x" can be an "s" to indicate a string, characters retrieved from the corresponding source operand are converted to character format; or an "x" can be a "c" to indicate a character, only one character is retrieved from the corresponding source operand, and is converted to character format.

Characters in mask which are not part of edit specifications are placed in output as literals. Mask can be a data name, or system register P1 - P8 only.

Continuous source contains the source data to be edited. The EDIT statement may contain up to six source operand. Mask must contain an edit specification for each source operand specified. Source can be a data name, a system register, or a literal.

END_PROC

The END_PROC verb identifies the last physical statement in a procedure definition. Control returns to the "calling" procedure and program execution continues with the statement following the "call" statement. The syntax for END_PROC is:

```
END_PROC;
```

An END_PROC statement is required as the last physical statement in every procedure. Accordingly, a procedure may contain only one END_PROC statement.

An END_PROC statement in a "called" procedure is equivalent to a RETURN statement. Further, an END_PROC statement in the highest level procedure of a program is equivalent to an EXIT statement.

EXIT

The EXIT verb is used to transfer program control to the Reception System. When EXIT executes, the currently running program is ended. The data in operand 1 is moved to SYS_RETURN_CODE, and control is returned to the Reception System.

The syntax for EXIT is:

```
EXIT return-code;
```

where "return-code" contains data to be moved to SYS_RETURN_CODE prior to transfer of control to the Reception System. A value of 0 indicates a normal return. A non-zero value indicates an error condition. Return_code can be a data name, system register, or a literal.

The EXIT verb is the normal way to end processing in a TBOL program. In the highest level procedure of a program a RETURN or an END_PROC is equivalent to an EXIT.

FETCH

The FETCH verb is used to retrieve an object from a host system or from the Reception System storage device stage. The object specified in operand 1 is retrieved from its present location and made available in the Reception System. If operand 2 is specified, the object's data segment is placed in the object's data segment is placed in the operand 2 field.

The syntax for FETCH is:

```
FETCH object_id [,field];
```

where "object_id" contains the object ID of the object to be located and retrieved. Object_id can be a data name, system register P1 - P8 only, or a literal.

In the FETCH syntax "field" contains the name of a field to hold the retrieved object's data segment. Field can be a data name, or a system register P1 - P8 only.

When an object might be required for subsequent processing, the field operand should not be specified in the FETCH statement. In that case, the FETCH will be an asynchronous task and the program will not experience a wait. The object is placed in the Reception System ready for use. The field operand is specified when an object is required to immediate use. Here, the FETCH is a synchronous task and the program may experience a wait. When the FETCH is completed, the program has access to the FETCHed object's data segment in the field operand.

FILL

The FILL verb is used to duplicate a string of characters repeatedly within a field. The character string pattern contained in operand 2 is duplicated repeatedly in operand 1 until the length of operand 1 is equal to the number specified in operand 3. The syntax for FILL is:

```
FILL output,pattern,length;
```

where "output" is the name of the field to be filled with the character string specified in "pattern". Output can be a data name or a system register P1 - P8 only, or a literal. Finally, "length" contains an integer number specifying the final length of output. Length can be a data name, system register or a literal.

FORMAT

The FORMAT verb is used to transfer a string of character data into variables defined in the DATA section of the program. The string of character data contained in operand 1 is transferred to DATA section variables using destination and length specification in the format map contained in operand 2. The FORMAT syntax is:

```
FORMAT source,map;
```

where "source" contains a string of character data to be transferred to DATA section variables, and can be a data name or system register P1 - P8 only.

Continuing, "map", on the other hand, contains a format map consisting of a destination/length specification for each field

of data to be transferred. Map is created with the MAKE_FORMAT verb prior to execution of the statement.

GOTO

The GOTO verb transfers control to another statement within the currently running procedure. Program execution continues at the statement with the label identifier specified as operand 1. The syntax for GOTO is:

```
GOTO label_id;
```

where "label_id" is a label identifier directly preceding a statement within the currently running procedure. A GOTO statement can be used to transfer control to another procedure. Transfer to another procedure is accomplished by using the target procedure name as the verb in a statement.

GOTO_DEPENDING_ON

The GOT_DEPENDING_ON verb transfers control to one of several other statements within the currently running procedure. Operand 1 contains a number, and is used as an index to select one of the label identifiers beginning with operand 2 in the statement. Program execution continues at the statement with the selected label identifier.

The syntax for GOTO_DEPENDING_ON is:

```
GOTO_DEPENDING_ON index,label_id [,label_id...];
```

where "index" is an integer number used to select one of the label identifiers in the statement as the point where program execution will continue. If index contains a 1, then program execution continues at the statement with the label identifier specified as operand 2. If index contains a 2, then program execution continues at the statement with the label identifier specified as operand 3. And so on. If there is no label_id operand corresponding to the value in index, then program execution continues with the statement following the GOTO_DEPENDING_ON statement. Index can be a data name or system register. Continuing, "label_id" is a label identifier directly preceding a statement within the currently running procedure. Up to 147 label_id operands may be specified in a GOTO_DEPENDING_ON statement.

A GOTO_DEPENDING_ON statement, however, cannot be used to transfer control to another procedure. Transfer to another procedure is done by using the target procedure name as the verb in a statement.

IF...THEN...ELSE

In this verb, the keyword IF directs the flow of program execution to one of two possible paths depending upon the evaluation of a boolean expression. The keyword IF is followed by a boolean expression. The boolean expression is always followed by

a THEN clause. The THEN clause may be followed by an ELSE clause. The boolean expression is evaluated to determine whether it is "true" or "false". If the expression is true, program execution continues with the THEN clause; the ELSE clause, if present, is- skipped. If the expression is false, the THEN clause is skipped; program execution continues with the statement following the clause or clauses.

The syntax for IF...THEN...ELSE is:

IF boolean THEN clause [ELSE clause];

where "boolean" is a boolean expression. Boolean can be a single relational expression or two or more relational expressions separated by the key words AND and OR. These relational expressions can be enclosed with parentheses, and then treated as a single relational expression separated from others with AND or OR. They are evaluated from left to right.

In the syntax, "clause" can be: a single statement, or a block of statements. Where clause is a block of statements, the block begins with the keyword DO and ends with the END verb. Further, Clause is always preceded by the keyword THEN or ELSE.

INSTR

The INSTR verb searches a character string to determine if a specific substring of characters is contained within it. The character string in operand 1 is searched for the first occurrence of the character string in operand 2. If a matching string is found in operand 1, an integer number specifying its starting position is placed in operand 3. If a matching string is not found, 0 is placed in operand 3.

The syntax for INSTR is: INSTR string,pattern,strt_pos;

INSTR string, pattern, strt_pos;

where "string" contains the character string to be searched. String can be a data name, system register P1 - P8 only, or a literal.

Continuing, "pattern" contains the character string pattern which may occur within the string operand, and can be a data name, system register P1 - P8 only, or a literal.

Finally, "strt_pos" is the name of the variable where the starting position (or o) is to be stored. Strt_pos can be a data name, or system register P1 - P8 only.

LENGTH

The LENGTH verb is used to determine the length of a specified variable. An integer number specifying the number of

characters in operand 1 is placed in operand 2. The syntax for LENGTH is:

```
LENGTH field,length;
```

where "field" contains the data whose length is to be determined. Field can be a data name, system register P1 - P8 only, or a literal.

Continuing, on the other hand, "length" is the name of the variable which is to contains the length of the field operand, and can be a data name, or a system register P1 - P8 only.

LINK

The LINK verb transfers control to another TBOL program. Program execution continues at the first statement in the program whose object ID is contained in operand 1. Up to eight parameters may be passed to the "called" program in operand 2 - 9. Control returns to the statement following the LINK statement when the "called" program performs an EXIT.

The syntax for LINK is:

```
LINK object_id [,parameter...];
```

where "object_id" contains the object ID of a TBOL program, and can be data name, system register P1 - P8, only or a literal.

Further, "parameter" contains parameter data for the program whose object ID is contained in operand 1. The contents of the parameter operand 2 through 9, if present, are placed in parameter registers P1 through P8. The number of parameter operand is placed in P0. P0 through P8 are accessible to the "called" program. Parameter can be a data name, system register, or a literal.

LOOKUP

The LOOKUP verb issued to search for an entry in a table of data contained in a character string. Operand2 contains a single character string consisting of a number of logical records of equal length. Each record consists of a fixed-length key field and a fixed-length data field. Operand 3 contains the record length.

Operand 1 contains a search key equal in length to the length of the key field. OPerand 2 in searched for a record with a key field equal to operand 1. If a record with a matching key is found, an integer number specifying its starting position is placed in operand 4. If a matching record is not found, 0 is placed in operand 4.

The syntax for LOOKUP is:

```
LOOKUP schkey,table,rcd_lth,result;
```

where "schkey" contains the key data of the desired record and can be a data name, system register or a litera. Further, "table" contains a character string consisting of a number of equal length logical records, and be a data name or system register P1 - P8 only. Yet further, "rcd)lth" contains an integer number equal to the length of a record in a table, and can be a data name, system register, or a literal. Finally, "result" is the name of the field to receive the result of the search. Result can be a data name, or a system register.

MAKE_FORMAT

The MAKE_FORMAT verb is used to create a format map for use with the FORMAT verb. From 1 to 255 destination/length specifications contained in operands (beginning in operand 2) are used to create a format map which is stored in operand 1. Operand 1 can then be specified as the map operand in a FORMAT statement.

The MAKE_FORMAT syntax is:

```
MAKE_FORMAT map,format[,format...];
```

where "map" is the name of the variable which is to contain the format map created with this statement. Map will be specified as an operand in a subsequent FORMAT statement to control the transfer of a string of character data to variables. Map can be either a data name or system register P1 - P8 only. Continuing, "format" contains a destination/length specification for one

logical field of a string of character data. From 1 to 255 format operands can be specified in this statement to create a format map. Each format operand controls the transfer of one logical field of data from a character string when the format map created in this statement is used in a subsequent FORMAT statement. In this arrangement, format can be a data name or a system register P1 - P8 only.

A destination/length specification in a format operand always contains a destination field name. The field name is followed by either one or two integer numbers controlling the length of the designation field data. The field name and numbers are separated by the colon character, e.g., destination:fix_lth:imbed_lth, or destination:fix_lth, or as destination::imbed_lth.

For this approach, "destination is a variable field name which will contain the logical field of data from the character string after the subsequent performance of the FORMAT verb. And, "fix_lth" is an integer number between 1 and 33767 specifying a fixed field length for destination. If fix_lth is not specified then 2 colon characters are used to separate destination from imbed_lth, showing that fix_lth has been omitted. In this case, the destination field length is controlled entirely by imbed_lth, which must be specified. If fix_lth is specified and imbed_lth is not, then fix_lth characters will be transferred to destination during the subsequent performance of the FORMAT verb.

Finally, if fix_lth is specified with imbed_lth, then destination will have a length of fix_lth after the transfer of data by the FORMAT verb.

Continuing, "imbed_lth" is an integer number, either 1 or 2 which specifies length of an imbedded length field that immediately precedes the logical field of data in the character string. The imbedded length field contains the length of the logical field of data immediately following. For example, 1 specifies a 1-character length field and 2 specified a 2-character length field.

If imbed_lth is not specified then the designation field length is controlled entirely by fix_lth, which must be specified. If imbed_lth is specified and fix_lth is not, then the number of characters transferred to destination from the character string is controlled by the number in the one or two-character length field which precedes the logical field of data. If imbed_lth is specified with fix_lth, then the number of characters transferred to destination from the character string is controlled by the-

number in the one or two-character length field which precedes the logical field of data. After the transfer of data, if the length of destination is not equal to fix_lth, then it is either truncated, or extended with blank characters as necessary.

MOVE

The move verb copies data from one or more source fields into an equal number of destination fields. The data contained in the operand 1 data structure field (or fields) replaces the contents of the operand 2 data structure field (or fields). Operand 1 data remains unchanged. Normally, the moved data is converted to the data type of the destination. If the key word ABS is included as operand 3, then data conversion does not take place.

The syntax for MOVE is:

```
MOVE source,destination[, ABS];
```

where "source" is the name of the data structure containing the data to be moved, and can be a data name, or a group data name, or system register, or a literal. Further "destination" is the name of the data structure field (or fields) to receive the source data, and can be a data name, or group data name, or a system register. Finally, "ABS" is a keyword specifying an absolute move; i.e., no data conversion takes place. However, data residing in an integer register will always be in binary integer; and data residing in a decimal register will always be in internal decimal format.

If the source operand is a group data name, then the destination operand must be a group data name. Further, data in all of the fields contained in the source data structure or array are

moved to the corresponding fields in the destination data structure or array.

MULTIPLY

The MULTIPLY verb multiplies two numbers. The number in operand 2 is multiplied by the number in operand 1. The number in operand 1 is unchanged. The number in operand 2 is replaced with the product of the two numbers. The syntax for MULTIPLY is:

```
MULTIPLY number1,number2;
```

where "number1" contains the first number factor for the multiply operation, and can be a data name, system register or literal; and "number2" contains the second number factor for the multiply operation. Following execution, the contents of number2 are overlaid with the resulting of the product. Number2 can be a data name, or a system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the add operation. Fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Negative results will contain a minus sign (-) in the left-most position.

NAVIGATE

The NAVIGATE verb is used to transfer control to the TBOL program logic associated with different page template objects. The external effect is the display of a new screen page. Operand 1 contains either a page template object ID, or a keyword representing a navigation target page. Control is returned to the Reception System where the necessary objects are acquired and made ready to continue the videotex session at the specified new page.

The syntax for NAVIGATE is:

```
NAVIGATE object_id;
```

where "object_id" contains the object ID of a target page template object, and can be a data name, register P1 - P8 only, or a literal.

NOTE

The NOTE verb returns the current position of the file pointer in a reception system file. Operand 1 contains the name of a file. An integer number specifying the current position of the file's pointer is returned in operand 2. The NOTE syntax is:

```
NOTE filename,position;
```

where "filename" contains the name of a reception system file. The name of the file must be a valid MS-DOS file specification;

e.g.[drive:][\path\]name[.extension]. Filename can be a data name, or a system register P1 - P8 only. Continuing, "position" is the name of the field to receive the current position of the file pointer for the file specified in filename. This will be an integer number equal to the numeric offset from the beginning of the file; a 10 in position means the file pointer is positioned at the 10th character position in the file. Position can be a data name, or system register.

OPEN

The OPEN verb is used to open a reception system file for file processing. The file named in operand 1 is opened for processing in the mode specified an operand 2. The syntax for OPEN is:

OPEN filename, INPUT:OUTPUT:I/O:APPEND:BINARY; where "filename" contains the name of the reception system file to be opened. As will be appreciated with this convention, the file name PRINTER specified the system printer. Otherwise, the name of the file must be a valid MS-DOS file specification; e.g.[drive:][\path\]name[.extension]. Filename can be a data name, or system register P1 - P8 only.

Further, "INPUT" is a keyword specifying that the file is to be opened for reading only; "OUTPUT" is a keyword specifying that the file is to be opened for writing only; "I/O" is a key word specifying that the file is to be opened for both reading and

writing; "APPEND" is a keyword specifying that the file is to be opened for writing, where new data is appended to existing data; and "BINARY" is a keyword specifying that the file is to be opened for both reading and writing. Where all file data is in binary format.

OPEN_WINDOW

The OPEN_WINDOW verb is used to open a window on the base screen. The window whose object ID is contained in operand 1 is opened. Program execution continues with the first statement of the program associated with the newly opened window object. The syntax for OPEN_WINDOW is:

```
OPEN_WINDOW window_id;
```

where "window_id" contains the object ID of the window to be opened on the base screen, and can be a data name, or system register P1 - P8 only or a literal .

After performance of the OPEN_WINDOW statement, program execution continues with the first statement of the window program; i.e., the program associated with the newly opened window object. A window program relinquishes control by performing a CLOSE_WINDOW. Although a window program cannot perform an OPEN_WINDOW, it can use CLOSE_WINDOW to close itself and open another window. This process can continue through several windows. FInally, when a window program performs a CLOSE_WINDOW

without opening a new window, program control does not work its way back through all the window programs. Instead, control returns to the non- window program which opened the first window. Program execution continues in that program with the statement following the OPEN_WINDOW statement. A window program can also close itself by performing one of the following verbs: NAVIGATE; or TRIGGER_FUNCTION. In such cases, control does not return to the program which opened the window.

OR

The OR verb performs a logical OR function on the bits of two data fields. The logical sum (OR) of the bits of operand 1 and operand 2 is placed in operand 2. Moving from left to right, the OR is applied to the corresponding bits of each field, bit by bit, ending with the last bit of the shorter field.

If the corresponding bits are 1 and 1, then the result bit is 1. If the corresponding bits are 1 and 0, or 0 and 1, then the result bit is 1. If the corresponding bits are 0 and 0, then the result bit is 0.

The data in operand 1 is left unchanged. The data in operand 2 is replaced by the result.

The syntax for OR is:

```
OR field1,field2;
```

where "field1" contains the first data field, and can be a data name, or system register I1 - I8 or P1 - P8 only, or a literal. Further, "field2" contains the second data field. The contents of field2 are overlaid by the result of the OR operation. Field2 can be a data name, or system register I1 - I8 or P1 - P8 only. As will be appreciated by those skilled in the art, the OR verb can be used to set a bit to 1.

POINT

The POINT verb is used to set the file pointer to a specified position in a reception system file. Operand 1 contains the name of a file. The file's pointer is set to the position specified by the integer number in operand 2. The POINT syntax is:

```
POINT filename,position;
```

where "filename" contains the name of a reception system file. The name of the file must be a valid MS-DOS file specification; e.g. [drive:][\path\]name[.extension]. File name can be a data name, or system register P1 - P8 only. Further, "position" contains an integer number equal to the desired position of the file pointer for the file specified in filename. A 10 in position means the file pointer will be positioned at the 10th character position in the file. Position can be a data name, or system register or literal.

POP

The POP verb transfers data from the top of the system stack to a variable field. The contents of operand 1 are replaced with data removed from the top of the system stack. The POP syntax is:

```
POP field;
```

where "field" is the name of the variable field to receive data from the stack, and can be a data name, or a system register.

PUSH

The PUSH verb transfers data from a variable field to the top of the system stack. The data contained in operand 1 is placed on the top of the system stack, "pushing down" the current contents of the stack. The contents of operand 1 remain unchanged. The PUSH syntax is:

```
PUSH field;
```

where "field" is the name of the variable field containing data to be "pushed" on the stack, and can be a data name, or a system register, or a literal.

READ

The READ verb is used to read data from a reception system file into a variable field. Operand 1 contains the name of a

file. Data is read from the file, beginning with the character position specified by the current contents of the file's pointer. Data read from the file replaces the contents of operand 2. Operand 3 may be present, containing an integer number specifying the number of characters to be read. For ASCII files, data is read from the file until the first end-of-line character (ASCII 13) is encountered. Or, if operand 3 is present, until the number of characters specified in operand 3 is read. For binary files, operand 3 is required to specify the length of the data to be read from the file.

The syntax for READ is:

```
READ filename,input [,length];
```

where "filename" contains the name of a reception system file, which must be a valid MS-DOS file specification, e.g. [drive:][\path\]name[.extension]. Filename can be a data name, or system register P1 - P8 only. Continuing, "input" is the name of the variable field to receive data read from the file, and can be a data name, or a system register P1 - P8 only. Finally, "length" contains an integer number. For ASCII files, length specifies the maximum number of characters to be read. For binary files, length specifies the length of the data to be read.

As will be appreciated by those skilled in the art, in order to perform a READ operation, a file must first be opened as INPUT or I/O before the READ operation can take place.

RECEIVE

The RECEIVE verb is used to access the expected reply to a message sent previously to a host system. Operand 1 contains the message ID of a message sent previously to a host system. The message reply from the host replaces the contents of operand 2. The RECEIVE syntax is:

```
RECEIVE msg_is,message;
```

where "msg_id" contains the ;message ID of a message sent previously to a host system, and can be a data name, or a system register P1 - P8 only. Further, "message" is the name of the variable field to receive the incoming message reply, and can be a data name, or a system register P1 - P8 only.

RELEASE

The RELEASE verb reclaims memory space in the reception system by deleting a block of data saved previously with the SAVE verb. The block of data named in operand 1 is deleted from memory.

The syntax for RELEASE is:

```
RELEASE block_name;
```

where "block_name" contains a block name used in some previously performed SAVE statement,and can be a literal.

RESTORE

The RESTORE verb is used to restore the previously saved contents of a block of variables. The block of data named in operand 1 replaces the contents of a block of variables, beginning with the variable named in operand 2. The RESTORE syntax is:

```
RESTORE block_name,field1;
```

where "block_name" contains a block name used in some previously performed SAVE statement, and can be a literal. Further, "field1" is the name of the first field or a data structure to receive data from the block specified in block_name. Field1 can be a data name, or a group data name.

RETURN

The RETURN verb is used to return control to the procedure which "called" the currently running procedure. Execution of the currently running procedure is ended. The data in operand 1 is moved to SYS_RETURN_CODE, and control is returned to the procedure which "called" the currently running procedure.

The RETURN syntax is:

RETURN return-code;

where "return-code" contains data to be moved to SYS_RETURN_CODE prior to transfer of control to the "calling" procedure, and can be a data name, or system register, or a literal. It should be noted that in the highest level procedure of a program, a RETURN or an END_PROC is equivalent to an EXIT.

SAVE

The SAVE verb is used to save the contents of a block of variables. Operand 1 contains a name to be assigned to the block of saved data. This name will be used later to restore the data. If operand 2 is specified without operand 3, then operand 2 may contain the name of a field, an array, or a data structure. In this case, the contents of the field; or the contents of all the elements in the array; or the contents of all the fields in the data structure are saved under the name specified in operand 1. If operand 2 and operand 3 are specified, then they both must contain a field name. In this case, the contents of all the fields, beginning with the field in operand 1 and ending with the field in operand 2, are saved under the name specified in operand 1.

The syntax for SAVE is:

```
SAVE block_name,name1 [,name2];
```

where "block_name" contains a block name to be assigned to the saved data, and will be used subsequently to restore the saved contents of the fields. Block_name can be a data name, system register P1 - P8 only, or a literal. Continuing, "name1" contains the name of a field, array, or data structure to be saved. If name2 is specified, name1 must contain a field name. Name1 can be a data name. Further, "name2" contains the last field name of a range of fields to be saved, and it can be a data name.

SEND

The SEND verb is used to transmit a message to a host system. The message text contained in operand 2 is transmitted from the reception system using a message header constructed from the data contained in operand 2. Operand 3, if present, indicates that an incoming response to the message is expected. The syntax for SEND is:

```
SEND message [,RESPONSE:TIMEOUT];
```

where "message" contains the outgoing message text (the header data for which has been placed in GEVs before SEND), and can be a data name, or a system register, or a literal. "RESPONSE" is a keyword indicating that a response to the message is expected. "TIMEOUT" is a parameter that sets the number of seconds for message time-out.

After performance of the SEND statement, the global external system variable SYS_LAST_MSG_ID contains a message ID number assigned to the outgoing message by the Reception System. This message ID number can be used later in a RECEIVE statement.

SET_ATTRIBUTE

The SET_ATTRIBUTE verb is used to set or change the color and input format attributes of a screen field. The characteristics of the screen field expressed as operand 1 are set or changed according to the specifications contained in operand 2. The syntax for SET_ATTRIBUTE is:

```
SET_ATTRIBUTE name, attr_list;
```

where "name" expresses the name of the field whose characteristics are to be set or changed. This is a partition external variable name, and if the name is expressed as a literal; e.g., "SET_ATTRIBUTE 1,...", then this is taken to mean that the attributes of the partition external variable &1 contains the name of the partition external variable whose attributes are to be set by this statement.

Further, "attr_list" is a literal character string containing a list of key words and values describing the desired attributes to be assigned to the field expressed in operand 1.

When SET_ATTRIBUTE is performed, existing field attributes remain in effect unless superseded by the attribute list contained in operand 2.

The attribute list operand literal is in the form: 'keyword[(values)][,keyword[(values)]...]'

It should also be noted that where key words and their associated values are: "DISPLAY", not user input data can be entered in a field with this attribute; "INPUT", a field with this attribute can receive user input data; "ALPHABETIC", an INPUT field with this attribute can receive any alphabetic character: A through A, and blank; "ALPHANUMERIC", an "INPUT", field with this attribute can receive any displayable character; "NUMERIC", an INPUT field with this attribute can receive any numeric character: 0 through 9, ($), (,), (.), and (-); "PASSWORD", an INPUT field with this attribute is intended for use as a password field. Any character entered by the user is displayed in the field as an asterisk (*); "ACTION", a field with this attribute is a TBOL "action" field; "COLOR(fg,bg)", where fg and bg are numeric values specifying the foreground and background colors of the field; "FORM(pattern)", where pattern specifies the input data format for this field. Pattern may contain "A", an alphabetic character of A through Z, which must be in this position; "a", an alphabetic character of A through Z, or a blank, which must be in this position; "N" a number character of 0 through 9, or ($), (,), (.), or (-) which must

be in this position; "n", a numeric character of 0 through 9, or ($), (,), (.), (-), or a blank may occupy this position; "X", any displayable character which must be in this position; and "x", any displayable character or a blank which must be in this position.

Any other character in the pattern is displayed in the field as a literal, and acts as an autoskip character at user input time. To include any of the pattern characters as literals in the pattern, they must be preceded by the backslash character. For example, to include the character "A: as a literal in a pattern it would code as "\A". To include the backslash character as a litera, it would code as "\\".

SET_CURSOR

The SET_CURSOR verb moves the cursor to the field specified as operand 1, itself specified as a field number. The syntax for the SET_CURSOR verb is:

SET_CURSOR [field number]

SET_FUNCTION

The SET_FUNCTION verb changes and/or filters a "logical function" process program. The syntax for SET_FUNCTION is: SET_FUNCTION function_id, status[,program_object_id [,state]]; where "function_id_ is the logical function" identifier; "status" is one of the following key words: "DISABLE"; "FILTER"; or

"ENABLE". DISABLE is used to de-activate "logical function". FILTER is used to execute the logic contained in program_object_id prior to executing the normal "logical function" process. It the logic contained in program_object_id returns a non-zero SYS_RETURN_CODE< the normal "logical function" process will not execute, otherwise, it begins. ENABLE is used to set "logical function" to normal default process.

Continuing, in the SET_FUNCTION syntax, "program_object_id" is the 13 byte object_id of the TBOL program, (conditional); and "state" is data to be passed to the "logical function" program. The data will reside in the P1 register when logic is executed, (optional).

SORT

The SORT verb is used to sort a range of variable fields into the sequence of the key contained in each field. Each variable field contains a record consisting of a fixed-length key field followed by a data field. The key field is the same length is each record. Operand 1 contains the name of the first field in the range of fields to be sorted; operand 2 contains the name of the last field. Operand 3 contains an integer number specifying the length of the key field contained in the beginning of each field. The fields in the range specified by operand 1 and operand 2 are sorted into the sequence of the key field.

The syntax for SORT is:

```
SORT field1,field2,key_lath;
```

where "field1" contains the first field name of the range of fields to be sorter, and can be a data name, or system register P1 - P8 only; "field2" contains the last field name of the range of fields to be sorted and can be a data name; or system register P1 - P8 only; and "key_lath" contains an integer number equal to the length of the key field contained in each field in the range. Key_lath can be a data name, or system register P1 - P8 only or a literal.

SOUND

The SOUND verb is used to produce a sound through the reception system speaker. A sound is produced of the pitch specified by operand 1, for the duration specified by operand 2, If operand 1 and operand 2 are not present, values from the most recently performed SOUND statement are used. The SOUND syntax is:

```
SOUND [pitch,duration];
```

where "pitch" is a numeric value in the range of 0 to 20,000 specifying the desired pitch of the sound. Pitch can be a data name, system register P1 - P8, or a literal; and "duration" is a numeric value in the range of 0 to 65,535 specifying the desired duration of the sound in increments of .1 seconds. Duration can be a data name, or system register P1 - P8 only or literal.

STRI

The STRING verb joins multiple character strings together with into one character string. Up to eight character strings, beginning with the character string contained in operand 1, are joined together sequentially. The resulting new character string replaces the contents of operand 1. The STRING syntax is:

STRING string1, [,string...];

where "string1" is empty, or contains the character string which will become the left-most portion of the new character string, and a data name, or a system register P1 - P8 only; "string" is empty, or contains the character string to be joined behind the character strings in preceding operands, and can be a data name, or system register P1 - P8 only or a literal.

SUBSTR

The SUBSTR verb is used to copy a substring of characters from a character string into a designated variable field. The character string containing the substring is in operand 1. Operand 3 contains an integer number equal to the position of the first character to be copied. Operand 4 contains an integer number equal to the number of characters to be copied. The specified substring is copied from the character string in operand 1 and replaces the contents of operand 2.

The syntax for SUBSTR is:

```
SUBSTR string,destination,strt_pos,length;
```

where "string" contains a character string, and can be a data name or system register P1 - P8 only, or a literal; "destination" is the name of the variable field to receive the substring copied from the string operand, and can be a data name, or system register P1 - P8 only, "strt,pos" contains an integer number specifying the position of the first character to be copied into the destination operand, and can be a data name, or system register or a literal; and "length" contains an integer number specifying the number of characters to be copied into the destination operand, and can be a data name, or system register or a literal.

In accordance with this arrangement, the SUBSTR operation does not take place if: if the length operand is 0, or if the strt_pos operand is greater than the length of the string operand.

SUBTRACT

The SUBTRACT verb subtracts one number from another. The number in operand 1 is subtracted from the number in operand 2. The number in operand 1 is unchanged. The number in operand 2 is replaced by the arithmetic difference between the two numbers. The syntax for SUBTRACT is:

```
SUBTRACT number1,number2;
```

where "number1" contains the number to be subtracted from number2, and can be a data name, or system register, or a literal; "number2" contains the second number. As noted, the contents of number2 are overlaid with the resulting difference. Number2 can be a data name, or system register.

TBOL will automatically perform data conversion when number1 is not the same data type as number2. Sometimes this will result in number2 having a different data type after the subtract operation. Fractions will be truncated after 13 decimal places, and whole numbers will not contain a decimal point. Further, negative results will contain a minus sign (-) in the left-most position.

TRANSFER

The TRANSFER verb transfers control to another TBOL program. Control however, does not return to the original program. Rather, program execution continues at the first statement in the program whose object ID is contained in operand 1. Up to eight parameters may be passed to the "called" program in operands 2 - 9. Control is transferred to the Reception System when the "called" program performs an EXIT.

The syntax for TRANSFER is:

```
TRANSFER object_id [,parameter...];
```

where, "object_id" contains the object ID of a TBOL program, and can be a data name, or system register P1 - P8 only, or a literal; "parameter" contains parameter data for the program whose object ID is contained in operand 1. The contents of the parameter operands 2 through 9, if present, are placed in parameter registers P1 through P8. The number of parameter operands is placed in P0. P0 through P8 are accessible to the "called" program. Parameter can be a data name, or system register, or a literal.

TRIGGER_FUNCTION

The TRIGGER_FUNCTION verb is designed to execute a "logical function". Its syntax is: TRIGGER_FUNCTION function_id; where "function_id" is the logical function" identifier. In accordance with the design of TRIGGER.FUNCTION, control may or may not be returned depending on the function requested.

UPPERCASE

The UPPERCASE verb converts lowercase alphabetic characters to uppercase alphabetic characters. Lowercase alphabetic characters (a - z) in the character string contained in operand 1 are converted to uppercase alphabetic characters (A - Z). The syntax for UPPERCASE is:

UPPERCASE string; where "string" contains a character string, and can be a data name, or a system register P1 - P8 only.

WAIT

The WAIT verb causes program control to be given to the REception System for the number of seconds defined in the parameter head. Control is given to the Reception System for one "time slice" and then returned to the currently running program.

The WAIT syntax is simply:

WAIT;seconds

WHILE...THEN

The key word WHEN causes a single statement or a block of statements to be executed repetitively while a specified boolean expression is true. The key word WHILE is followed by a boolean expression. The boolean expression is always followed by a THEN clause. The boolean expression is evaluated to determine whether it is "true" or "false". If the expression is true, the THEN clause is executed and the expression is evaluated again. If the expression is false, program execution continues with the statement following the THEN clause.

The syntax for WHILE...THEN is:

WHILE boolean THEN clause;

where "boolean is a boolean expression, which can be a single relational expression, where a relational expression consists of

two operands separated by a relational operator such as (=), (<>), (<), (>), (<=), or (=>), or two or more relational expressions separated by the key words AND or OR. These relational expressions can be enclosed with parentheses, and then treated as a single relational expression separated from others with and or OR. Further, they are evaluated from left to right. Continuing, with the syntax for WHILE...THEN, "clause" can be either a single statement, a block of statements, where the block begins with the key word GO and ends with the END verb.

When character strings of unequal length are compared lexicographically, the longer string is truncated to the length of the shorter string before the comparison. If the shorter string compares "high", then the longer string is "lower". For example: When comparing "GG" to "H", "GG" is valued as less than"H". If the shorter string compares "low" or "equal", then the longer string is "high". For example: When comparing "TO" to "TOO", "TO" is less than "TOO".

In this regard, truncation is done outside of the operands, which the operands remaining the same length after the evaluation.

WRITE

WRITE is the verb used to write records to a file. The syntax for WRITE is:

WRITE filename , output_area [, key];

where "filename" is the name of the file that the record is to be written to, and can be a field_id, array_id(subscript), partition_external_id, global_external_id, or a literal; "output_area" is the name of the area from which the record will be created, and can be a field_id, array_id(subscript), partition_external_id or a global_external_id; and "length" specifies either the maximum number of characters to be read from an ASCII file, or the length of data to be read from a binary file. The file must have been previously opened as OUTPUT, APPEND, or I/O.

GLOBAL EXTERNAL SYSTEM VARIABLES

In accordance with the design of TBOL, names have been assigned to the TBOL system variables in the global external variable (GEV) data structure. The names of GEVs are assigned in DEFINE statements as described above and in the file TBOL.SYS. There are a total of 32,000 GEVs. The first 256 GEVs are reserved for the system, and the remaining 31,744 are assigned as application variables, and are application-specific. Since system variables referenced by TBOL interpreter __ as global variables and are ascii strings, a system variable table is constructed so that reception system native code can access them as binary integer. An adaptation of this table from the source code file "\rs\rsk\c\sysvar.c" is shown in Table 1.

TABLE 1

SYSTEM GLOBAL EXTERNAL VARIABLES

System Variable Name GEV# Description /

Sys_rtn_code; / 0001 API instr. return code./
Sys_api_event; / 0002 API event: post,pre,init
or sel/ Sys_logical_key; / 0003 Current logical key.
/ Sys_last_msg_id; / 0004 Last message id. /
Sys_tone_pulse; / 0005 Phone type pulse/tone. /
Sys_line_status; / 0006 Line connection status. /
Sys_keyword; / 0007 Keyword flag. /
Sys_automatic_uppercase; / 0008 Auto uppercase. /
Sys_scroll_increment; / 0009 Scroll increment. /
Sys_current_field; / 0010 Current field. /
Sys_date; / 0011 system date. / Sys_time;
/ 0012 system time. / Sys_current_page; / 0013
current page. / Sys_selected_obj_id; / 0014 sel object
id. / Sys_navigate_obj_id; / 0015 nav object id. /
Sys_cursor_row; / 0016 cursor row position. /
Sys_cursor_col; / 0017 cursor col position. /
Sys_path ; / 0018 user personal path table.
/ Sys_ttx_phone; / 0019 dial trintex phone #. /
Sys_total_pages; / 0020 total pages in page set.

/ Sys_page_number; / 0021 Curr. page of of n pages.
/ Sys_base_obj_id; / 0022 curr. base page object-
id./ Sys_window_id; / 0023 Curr. window
object-id. / Sys_path_ptr; / 0024 Curr. path
location. / Sys_keywords; / 0025 Keyword list. /
Sys_current_cursor_pos; / 0026 Curr. cursor position. /
Sys_current_background_color; / 0027 Curr background color. /
Sys_current_foreground_color; / 0028 Curr foreground color. /
Sys_hardware_status; / 0029 nature of hard error. /
Sys_nocomm; / 0030 send:don't send to S1. /
Sys_um_dia_header; / 0031 header for unsolicited
msg. / Sys_um_message_text; / 0032 text for unsolicited
msg. / Sys_ca_error_track_info; / 0033 error tracking data.
/ Sys_assisant_current_info; / 0034 curr. context info. /
Sys_screen_data_table; / 0035 data table for copy and
file. / Sys_ad_list; / 0036 Pointer to AD
list. / Sys_current_keyword; / 0037 Pointer to
current keyword. / Sys_previous_keyword; / 0038
Pointer to previous keyword. / Sys_guide; /
0039 Guide. / Sys_previous_menu; / 0040 Prev
menu object-id. / Sys_previous_seen_menu; / 0041 Prev
seen menu obj-id. / Sys_scan_list; / 0042
Pointer to scan list. / Sys_scan_list_pointer; / 0043
User scan list pointer. / Sys_path_name; / 0044
Pointer to path name. / Sys_navigate_keyword; / 0045
Navigate to keyword. / Sys_keyword_table; / 0046

Sys_keyword_disp; / 0047

Sys_keyword_table_entry_length; / 0048

Sys_keyword_length; / 0049

Sys_ext_table; / 0050 Sys_data_collect ;

/ 0051 Indicates Tracking status Sys_fm0_txhdr;

/ 52 / DIA message header Sys_fm0_txdid; / 53

/ " Sys_fm0_txrid; / 54 / "

Sys_fm4_txhdr; / 55 / "

Sys_fm4_txuseid; / 56 / Sys_fm4_txcorid;

/ 57 / Sys_fm64_txhdr; / 58 /

Sys_fm64_txdata; / 59 / Sys_fm0_rxhdr;

/ 60 / Sys_fm4_rxhdr; / 61 /

Sys_fm4_rxuseid; / 62 / Sys_fm4_rxcorid;

/ 63 / Sys_fm64_rxhdr; / 64 /

Sys_fm64_rxdata; / 65 / Sys_surrogate;

/ 66 , md/ Sys_leave; / 67 , md/

Sys_return; / 68 , md/ Sys_int_regs;

/ 69 , md,area for int save stack/ Sys_ttx_help_id;

/ 70 , md,id of system help window/ Sys_selector_data;

/ 71 , md / Sys_selector_path; / 72 , md/

Sys_logical_event; / 73 , am/ Sys_user_id;

/ 74 , mg/ Sys_help_appl; / 75 , md/

Sys_help_hub_appl_pto; / 76 , md/

Sys_access_key_obj_id; / 77 , lw,bi/ Sys_word_wrap=1;

/ 78 , Sys_messaging_status; / 79 , Sys_version;

/ 80 , Sys_leader_ad_id; / 81 ,

Sys_baud_rate; / 82, Sys_com_port;

```
/ 8      Sys_obj_header ;                 / 84,
Sys_session_status;            / 85,      Systbl sys_var_table [] =
/ NA      Defne     system     variable    table.     &Sys_rtn_code,
INTLEN,                 SYS_INT_TYPE,                 &Sys_api_event,
INTLEN,                SYS_INT_TYPE,               &Sys_logical_key,
INTLEN,                SYS_INT_TYPE,               &Sys_last_msg_id,
INTLEN,                 SYS_INT_TYPE,               &Sys_tone_pulse,
INTLEN,                SYS_INT_TYPE,               &Sys_line_status,
INTLEN,                  SYS_INT_TYPE,                  &Sys_keyword,
INTLEN,            SYS_INT_TYPE,        &Sys_automatic_uppercase,
INTLEN,             SYS_INT_TYPE,          &Sys_scroll_increment,
INTLEN,              SYS_INT_TYPE,             &Sys_current_field,
INTLEN,  SYS_INT_TYPE, &(unsigned int)Sys_date,                    0,
SYS_STR_TYPE,     &(unsigned    int)Sys_time,                      0,
SYS_STR_TYPE,       &Sys_current_page,                             0,
SYS_INT_TYPE,     &(unsigned    int)Sys_selected_obj_id,           0,
SYS_STR_TYPE,     &(unsigned    int)Sys_navigate_obj_id,           0,
SYS_STR_TYPE,       &Sys_cursor_row,                               0,
SYS_INT_TYPE,       &Sys_cursor_col,                               0,
SYS_INT_TYPE,     &(unsigned    int)Sys_path,                      0,
SYS_STR_TYPE,     &(unsigned    int)Sys_ttx_phone,                 0,
SYS_STR_TYPE,    &Sys_total_pages,                            INTLEN,
SYS_INT_TYPE,    &Sys_page_number,                            INTLEN,
SYS_INT_TYPE,     &(unsigned    int)Sys_base_obj_id,               0,
SYS_STR_TYPE,     &(unsigned    int)Sys_window_id,                 0,
SYS_STR_TYPE,    &Sys_path_ptr,                               INTLEN,
SYS_INT_TYPE,     &(unsigned    int)Sys_keywords,                  0,
```

```
SYS_STR_TYPE,     &Sys_current_cursor_pos,                          INTLEN,
SYS_INT_TYPE,     &Sys_current_background_color,                    INTLEN,
SYS_INT_TYPE,     &Sys_current_foreground_color,                    INTLEN,
SYS_INT_TYPE,     &Sys_hardware_status,                             INTLEN,
SYS_INT_TYPE,     &Sys_nocomm,                                      INTLEN,
SYS_INT_TYPE,      &(unsigned    int)Sys_um_dia_header,                 0,
SYS_STR_TYPE,      &(unsigned    int)Sys_um_message_text,               0,
SYS_STR_TYPE,      &(unsigned    int)Sys_ca_error_track_info,           0,
SYS_STR_TYPE,     &(unsigned    int)Sys_assisant_current_info           0,
SYS_STR_TYPE,     &(unsigned    int)Sys_screen_data_table,              0,
SYS_STR_TYPE,     &(unsigned    int)Sys_ad_list,                        0,
SYS_STR_TYPE,     &(unsigned    int)Sys_current_keyword,                0,
SYS_STR_TYPE,     &(unsigned    int)Sys_previous_keyword,               0,
SYS_STR_TYPE,     &(unsigned    int)Sys_guide,                          0,
SYS_STR_TYPE,     &(unsigned    int)Sys_previous_menu,                  0,
SYS_STR_TYPE,     &(unsigned    int)Sys_previous_seen_menu,             0,
SYS_STR_TYPE,     &(unsigned    int)Sys_scan_list,                      0,
SYS_STR_TYPE,     &(unsigned    int)Sys_scan_list_pointer,              0,
SYS_STR_TYPE,     &(unsigned    int)Sys_path_name,                      0,
SYS_STR_TYPE,     &(unsigned    int)Sys_navigate_keyword,               0,
SYS_STR_TYPE,     &(unsigned    int)Sys_keyword_table,                  0,
SYS_STR_TYPE,    &Sys_keyword_disp,                                 INTLEN,
SYS_INT_TYPE,    &Sys_keyword_table_entry_length,                   INTLEN,
SYS_INT_TYPE,     &Sys_keyword_length,                              INTLEN,
SYS_INT_TYPE,     &(unsigned    int)Sys_ext_table,                      0,
SYS_STR_TYPE, &()Sys_data_collect, &(unsigned int) Sys_fm0_txhdr,
0  ,  SYS_STR_TYPE, &(unsigned int) Sys_fm0_txdid,                  0  ,
```

```
SYS_STR_TYPE,   &(unsigned   int)   Sys_fm0_txrid,            0,
SYS_STR_TYPE,        &(unsigned int) Sys_fm4_txhdr,           0,
SYS_STR_TYPE,     &(unsigned   int)   Sys_fm4_txuseid,        0,
SYS_STR_TYPE,        &(unsigned int) Sys_fm4_txcorid,         0,
SYS_STR_TYPE,        &(unsigned int) Sys_fm64_txhdr,          0,
SYS_STR_TYPE,     &(unsigned   int)   Sys_fm64_txdata,        0,
SYS_STR_TYPE,        &(unsigned int) Sys_fm0_rxhdr,           0,
SYS_STR_TYPE,     &(unsigned   int)   Sys_fm4_rxhdr,          0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_fm4_rxuseid,          0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_fm4_rxcorid,          0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_fm64_rxhdr,           0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_fm64_rxdata,          0,
SYS_STR_TYPE,   &Sys_surrogate,                          INTLEN,
SYS_INT_TYPE,   &(unsigned   int)   Sys_leave,                0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_return,               0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_int_regs,             0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_ttx_help_id,          0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_selector_data,        0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_selector_path,        0,
SYS_STR_TYPE,   &Sys_logical_event,                      INTLEN,
SYS_INT_TYPE,   &(unsigned   int)   Sys_user_id,              0,
SYS_STR_TYPE,   &Sys_help_appl,                          INTLEN,
SYS_INT_TYPE,   &(unsigned   int)   Sys_help_hub_appl_pto,    0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_access_key_obj_id,    0,
SYS_STR_TYPE,     &Sys_word_wrap,                             1,
SYS_INT_TYPE,    &(unsigned    int)Sys_messaging_status,      0,
SYS_STR_TYPE,   &(unsigned   int)   Sys_version,              0,
```

```
SYS_STR_TYPE,   &(unsigned   int)   Sys_leader_ad_id,          0,
SYS_STR_TYPE,   &Sys_baud_rate,                            INTLEN,
SYS_INT_TYPE,   &Sys_com_port,                             INTLEN,
SYS_INT_TYPE,     &Sys_obj_header,                             0,
SYS_STR_TYPE,/RDC                           &Sys_session_status,
INTLEN, SYS_INT_TYPE,
```

Table 2

TBOL VERBS BY FUNCTIONAL CATEGORY

DATA PROCESSING

	LOOKUP	SORT ADD
MAKE_FORMAT	STRING AND	MOVE
SUBSTR CLEAR	MULTIPLY	SUBTRACT DIVIDE
OR	UPPERCASE	
EDIT	POP	SAVE
FILL	PUSH	XOR
FORMAT	RELEASE	
INSTR	RESTORE	
LENGTH		

PROGRAM FLOW

CLOSE_WINDOW	NAVIGATE	SYNC_RELEASE
ERROR	OPEN_WINDOW	TRANSFER
EXIT	RETURN	TRIGGER_FUNCTION
GOTO	SET_FUNCTION	WAIT
GOTO_DEPENDING_ON		WHILE...THEN LINK
IF...THEN...ELSE		

COMMUNICATIONS

CONNECT	RECEIVE

DELETE

DISCONNECT

FILE MANAGEMENT CLOSE

NOTE

OPEN

POINT

READ

SEND

WRITE

SCREEN MANAGEMENT

DEFINE_FIELD

SET_ATTRIBUTE

SET _CURSOR

SOUND

REFRESH FILE_SCREEN

OBJECT MANAGEMENT

FETCH

PROGRAM STRUCTURE

DO...END

END_PROC

RECEPTION SYSTEM OPERATION

RS 400 of computer system network 10 uses software called native code modules (to be described below) to enable the user to select options and functions presented on the monitor of personal computer 405, to execute partitioned applications and to process user created events, enabling the partitioned application to interact with interactive system 10. Through this interaction, the user is able to input data into fields provided as part of the display, or may individually select choices causing a standard or personalized page to be built (as explained below) for display on the monitor of personal computer 405. Such inputs will cause RS 400 to interpret events and trigger pre-processors or post-processors, retrieve specified objects, communicate with system components, control user options, cause the display of advertisements on a page, open or close window partitions to provide additional navigation possibilities, and collect and report data about events, including certain types of objects processed. For example, the user may select a particular option, such as opening or closing window partition 275, which is present on the monitor and follow the selection with a completion keystroke, such as ENTER. When the completion keystroke is made, the selection is translated into a logical event that triggers the execution of a post-processor, (i.e., a partitioned application program object) to process the contents of the field.

Functions supporting the user-partitioned application interface can be performed using the command bar 290, or its

equivalent using pull down windows or an overlapping cascade of windows. These functions can be implemented as part of the RS native functions or can be treated as another partition(s) defined for every page for which an appropriate set of supporting objects exist and remain resident at RS 400. If the functions are part of RS 400, they can be altered or extended by verbs defined in the RS virtual machine that permit the execution of program objects to be triggered when certain functions are called, providing maximum flexibility.

To explain the functions the use of a command bar is assumed. The command bar 290 is shown in Figures 3(a) and 3(b) includes a NEXT command 291, a BACK command 292, a PATH command 293, a MENU command 294, and ACTION command 295, a JUMP command 296, a HELP command 297, and an EXIT command 298.

NEXT command 291 causes the next page in the current pageset to be built. If the last page of a pageset has already been reached, NEXT command 291 is disabled by RS 400, avoiding the presentation of an invalid option.

BACK command 292 causes the previous page of the current pageset to be built. If the present page is the first in the pageset, BACK command 292 is disabled, since it is not a valid option.

A filter program can be attached to both the NEXT or BACK functions to modify their implicit sequential nature based upon the value of the occurrence in the object set id.

PATH command 292 causes the next page to be built and displayed from a list of pages that the user has entered, starting from the fist entry for every new session.[p29]

MENU command 294 causes the page presenting the previous set of choices to be rebuilt.

ACTION command 295 initiates an application dependent operation such as causing a new application partition to be interpreted, a window partition 275 to be opened and enables the user to input any information required which may result in a transaction or selection of another window or page.

JUMP command 296 causes window partition 275 to be opened, allowing the user to input a JUMPWORD or to specify one from an index that may be selected for display.

HELP command 297 causes a new application partition to be interpreted such as a HELP window partaining to where the cursor is positioned to be displayed in order to assist the user regarding the present page, a particular partition, or a field in a page element.

EXIT command 298 causes a LOGOFF PTO to be built.

NAVIGATION INTERFACE

Continuing, as a further feature, the method aspect of the invention includes an improved procedure for searching and retrieving applications from the store of applications distributed throughout network 10; e.g., server 205, cache/concentrator 302 and the RS 400 units, as described above. More specifically, the procedure features techniques for reducing the demand on the server 205 for searching, and retrieving applications for display at monitor 414 by using pre-created search tables which represent subsets of the information on the network identified by page template objects and object ids so that in accordance with the procedure, the tables and associated objects can then be provided to and searched at the requesting RS 400, thus relieving server 205 from that responsibility.

In conventional time-sharing networks that support large In conventional databases, the host receives user requests for a data records; locating them; and transmitting them back to the users. Accordingly, the host is obliged to undertake the data processing responsibility necessary to provide the information, and, as noted earlier, where large numbers of users are to be served, the many user requests may bottleneck at the host, taxing resources and leading to response slowdown.

Further, users have experienced difficulty in searching data bases maintained on conventional time-sharing networks. For example, difficulties have resulted from the complex and varied way previously known database suppliers have organized and presented their information. Particularly, some database providers require searching be done only in selected fields of the data base, thus requiring the user to be fully familiar with the record structure. Others have organized their databases on hierarchial structures which require the user understand the way the records are grouped. Still further, yet other database suppliers rely upon keyword indices to facilitate searching of their records, thus requiring the user to be knowledgeable regarding the particular keywords used by the database provider.

The method aspect of the present invention, however, serves to avoid such difficulties. In the preferred embodiment, the method aspect of the present invention includes procedures for creating preliminary searches which represent subsets of the network applications that the users are believed likely to investigate. Particularly, in accordance with these procedures, for the active applications available on network 10, a library of tables is prepared, within each of which the tables a plurality of so called "JUMPwords" are provided that are correlated with page template objects and object ids associated with respective network applications. In the preferred embodiment, approximately 1,000

tables are uses, each having approximately 10 to 20 jumpwords each to abstract the applications on the network. Further, each table is associated with a code in the form of a character string mnemonic so that on entry of a character string at an RS 400, an appropriate table will be selected at server 205, which represents a subset of the objects and associated applications on the network, and that table will then be presented to the user's RS 400, where the RS 400 can provide the data processing required to present the potentially relevant jumpwords, objects and associated applications to the user for further review and determination as to whether more searching is required. This relieves the demand on server 205 and thereby permits it to be less complex and costly, and further, reduces the likelihood of overtaxing that may reduce network response time.

As a further feature of this procedure, the library of jumpwords and their associated PTOs may be generated by a plurality of operation which appear at the user's screen as different search techniques. This permits the user to select a search technique he is most comfortable with, to thereby expedite his inquiry.

More particularly, in accordance with the invention, the user is allowed to evoke the procedure by calling up a variety of

operation. The various operations have different names and seemingly present different search strategies to the user. Specifically, the user may initiate the procedure by invoking a so called "Jump" operation with the selection of a Jump command at RS 400. Thereafter, when prompted, the user may enter a word of the user's choosing at screen 414 relating to the matter he is interested in locating; i.e., a subject matter search of the network applications. Additionally, the users my evoke the procedure by alternatively calling up an operation termed "Index" by selecting the Index command, which command in response presents the user with an alphabetical listing of jumpwords from which the user can select; i.e., an alphabetical search of the network applications. Further, the user may evoke the procedure by initiating an operation termed "Guide" by selecting that command, which, thereafter, provides the user with a series of graphic displays that presents a physical description of the network applications; e.g., department floor plan for a store the user may be electronically shopping in. Still further, the user may evoke the procedures by initiating an operation termed "Directory" by selection of that command, which then presents the applications available on the network as a series of hierarchial menus which categorically arrange the content of the network information.

In accordance with the invention, this ability to convert these apparently different search strategies in a single procedure for accessing pre-created library tables is accomplished by

translating the procedural elements of the different search techniques by means of a conversion table into a single set of procedures that will produce a mnemonic which can be used to select the appropriate library table and associated jumpwords, PTO and object ids. Thus, while the search techniques may appear different to the user, and in fact accommodate the user's preferences and sophistication level, they nonetheless evoke the same efficient procedure of relying upon pre-created searches which identify related application PTOs and object ids so that the table and objects may be collected and presented at the user's RS 400 where they can be processed, thus relieving server 205.

A more detailed understanding of the procedure may be had upon a reading of the following description and review of accompanying Fig. 11 which presents a flow diagram of the search procedure.

To select a particular partitioned application from among thousands of such applications residing either at the RS 400 or within delivery system 20, the present invention avoids the need for a user to know or understand, prior to a search, the organization of such partitioned applications and the query techniques necessary to access them. This is accomplished using a collection of related commands, as described below.

The JUMP command 296, a flowchart for the execution of which is shown in Figure, is selected by the user from command bar 290,

which selection causes window partition 275 to open. At this point, the user may, in the preferred embodiment, select from a variety of options displayed on the monitor of personal computer in window partition 275. Among these options, the user may either select a DIRECTORY command, an INDEX command, a GUIDE command, or input into display field 270 within window partition 275 a "best guess" of the mneumonic character string that is assigned to a desired partitioned application (e.g., the user may input such english words as "news," "pet food," "games," etcetera). The JUMP command 296 will be discussed first, as it is related to and supports user learning in combination with the INDEX command and the DIRECTORY command , in ways that will become clear through discussion of the operation of the JUMP command 296.

The character string entered by the user in display field 270 of JUMP command 296 window partition 275 is searched by RS 400 native code (discussed below) against tables of keywords (not shown) that RS 400 requests from high function host 210. High function host 210 maintains a list of unique alphanumeric strings 13 character for PTOs and their associated PTOid's. The tables requested from high function host 210 are constructed by searching for keywords using the first characters of the string entered by the user, until a table of keywords within an arbitrary range of the first characters of the string is produced (in the preferred embodiment, considerations of line speed and usage have dictated that the table be 12 keywords). The table of keywords

give the closest approximations to the character string entered by the user. The table is then sent to RS 400 as message data.

If the string entered by the user is a unique keyword existing in the table of keywords, and is thus associated with a specific PTO, RS 400 fetches and displays the partitioned applications that are to be processed and built according to the page composition dictated by the target PTO as processed by the native code.

If the string entered by the user does not specify a unique PTO, RS 400 presents the user with the option to display the table of keywords as initialized, cursorable selector fields in an INDEX page. The user may then move the cursor to the nearest approximation of the nmeuonic he originally selected, and trigger navigation to the PTO associated with that keyword (by, in the preferred embodiment, pressing the "Return" key). Navigation is executed as described in the section of this specification entitled "RS NATIVE CODE."

If, instead of invoking the JUMP command 296, the user instead selects the INDEX command , the RS 400 will retrieve the keyword table residing in either the cache or stage of the local store 472, and will again build a page with initialized, cursorable fields of keywords. The table fetched upon invoking the INDEX command will be comprised of alphabetic keywords that occur within an arbitrary range of the keywords associated with

the PTO from which the user invoked the INDEX command . As discussed above, the user may select to navigate to any of this range of PTOs through selecting the relevant keyword from the display.

The DIRECTORY command will cause RS 400 to fetch from interactive system 10 a table of keywords, grouped by subject, to which the PTO of the current partitioned application (as specified by the object set field 630 of the current PEO) belongs. The RS 400 displays to the user these keywords together with descriptive statements about the application associated with the keywords. The user who can, in the manner previously discussed with regard to the INDEX command , select from and navigate to the PTOs of keywords which are related to the present pageset by subject.

The GUIDE command provides a navigation method related to a hierarchical orgainzation of applications which are described in a series of menus. Such menus, which use overlaying windows of subject areas, are well known to the art, and are expressed in a different manner in, for example, software for the Apple MacIntosh. The GUIDE command makes use of the keyword segment which describes the location of the PTO in a hierarchy (referred to, in the preferred embodiment, as the "BFD," or Building-Floor-Department) as well as the JUMPword character string. The BFD describes the set of menus that should be displayed in windows on the screen in a series of pop-up menus. GUIDE command may be

invoked by requesting such from a menu presented to the user, or by selecting the MENU command when the stack of PTOid's for MENU is empty.

The PATH command is a list of user pre-selected keywords that may be presented to user by electing the VIEWpath command. As in the INDEX command the PATH command presents the pre-selected keywords as initialized, cursorable fields, which the user may select from using the tab and return keys on the keyboard of the personal computer.

Collectively, the JUMP command 296, the INDEX command , the DIRECTORY command , the GUIDE command , and the PATH command enable the user to quickly and easily ascertain the "location" of either the partitioned application presently displayed or the "location" of a desired partitioned application. "Location," as used in reference to the preferred embodiment of the invention, means the specific relationships that a particular partitioned application bears to other such applications, and the method for selecting particular partitioned applications from such relationships. The techniques for querying a database of objects, embodied in the present invention, is an advance over the prior art, insofar as no foreknowledge of either database structure or query technique or syntax is necessary. The structure and search techniques are made manifest to the user in his use of the above commands.

RS APPLICATION PROTOCOL

RS protocol defines the way the RS supports user - application conversation (input and output) and the way RS 400 processes a partitioned application. Partitioned applications are constructed knowing that this protocol will be supported unless modified by the application. The protocol is illustrated FIG. 6. The boxes in FIG. 6 identify processing states that the RS passes through and the arrows indicate the transitions permitted between the various states and are annotated with the reason for the transition.

The various states are: (A) Initialize RS, (B) Process Objects, (C) Interpretively Execute Pre-processors, (D) Wait for Event, (E) Process Event, and (F) Interpretively Execute Function Extension and/or Post-processors.

The transitions between states are: (1a) Logon PTO-id, (1b) Object-id, (2) Trigger PDO-id & return, (3) PPT or Window Stack Processing complete, (4) Event occurrence, and (5) Trigger PDO-id and return.

Transition (1a) from Initialize RS (A) to Process Objects (B) occurs when an initialization routine passes the object-id of the logon PTO to object interpreter 435, when the service is first invoked. Transition (1b) from Process Event (E) to Process

Objects (B) occurs whenever a navigation class event causes a new PTO object-id to be passed to object interpreter 435; or when a open window event (verb or function key) occurs passing a window Object-id to the object interpreter 435; or a close window event (verb or function key) occurs causing the current topmost window to be closed.

While in the process object state, object interpreter 435 will request any objects that are identified by external references in call segments. Objects are processed by parsing and interpreting the object and its segments according to the specific object architecture. As object interpreter 435 processes objects, it builds a linked list structure called a Page Processing Table (PPT), shown in figure 10, to reflect the structure of the page, each page partition, PEOs required, Program Data Objects (PDOs) required and each window object that could be called. Object interpreter 435 requests all objects required to build a page except objects that could be called as the result of some event, such as a HELP window object.

Transition (2) from Process Objects (B) to Interpretively Execute Pre-processors (C) occurs when the object interpreter 435

determines that a pre-processor is to be triggered. Object processor 458 then passes the object-id of the (PDO) to the TBOL interpreter 438. TBOL interpreter 438 uses the RS virtual machine to interpretively execute the PDO. The PDO can represent

either a selector or an initializer. When execution is complete, a transition automatically occurs back to Process Objects (B).

Selectors are used to dynamically link and load other objects such as PEOs or other PDOs based upon parameters that they are passed when they are called. Such parameters are specified in call segments or selector segments. This feature enables RS 400 to conditionally deliver information to the user base upon pre-determined parameters, such as his personal demographics or locale. For example, the parameters specified may be the transaction codes required to retrieve the the user's age, sex, and personal interest codes from records contained in user profiles stored at the switch/file server layer 200.

Initializers are used to set up the application processing environment for a partitioned application and determine what events RS 400 may respond to and what the action will be.

Transition (3) from Process Objects (B) to Wait for Event (D) occurs when object interpreter 435 is finished processing objects associated with the page currently being built or opening or closing a window on a page. In the Wait for Event state (D), input manager accepts user inputs. All keystrokes are mapped from their physical codes to logical keystrokes by the Keyboard Manager 434, representing keystrokes recognized by the RS virtual machine.

When the cursor is located in a field of a page element, keystrokes are mapped to the field and the PEV specified in the PEO field definition segment by the cooperative action of input manager 452 and display manager 461. Certain inputs, such as RETURN or mouse clicks in particular fields, are mapped to logical events by keyboard manager 434, which are called completion (or commit) events. Completion events signify the completion of some selection or specification process associated with the partitioned application and trigger a partition level and/or page level post-processor to process the "action" parameters associated with the user's selection and commit event.

Such parameters are associated with each possible choice or input, and are set up by the earlier interpretive execution of an initializer pre-processor in state (C). Parameters usually specify actions to perform a calculation such as the balance due on an order of several items with various prices using sales tax for the user's location, navigate to PTO-id, open window WO-id or close window. Actions parameters that involve the specification of a page or window object will result in transition (1b) to the Process Objects (B) state after the post-processor is invoked as explained below.

Function keys are used to specify one or more functions which are called when the user strikes these keys. Function keys can include the occurrence of logical events, as explained above. Additionally, certain functions may be "filtered", that is,

extended or altered by SET_FUNCTION or TRIGGER_FUNCTION verbs recognized by the RS virtual machine. Function keys cause the PDO specified as a parameter of the verb to be interpretively executed whenever that function is called. Applications use this technique to modify or extend the functions provided by the RS.

Transition (5) from Process Event (E) to Interpretively Execute Pre-processors (F) occurs when Process Event State determines that a post-processor or function extension PDO is to be triggered. The object-id of the PDO is then passed to the id of the Program Data Object (PDO) to the TBOL interpreter 438. The TBOL interpreter 438 uses the RS virtual machine to interpretively execute the PDO. When execution is complete a transition automatically occurs back to Process Event (E).

RECEPTION SYSTEM SOFTWARE

The reception system 400 software is the interface between the user of personal computer 405 and interactive network 10. The object of reception system software are to minimize mainframe processing, minimize transmission across the network, and support application extendability and portability.

RS 400 software is composed of several layers, as shown in FIG. 7: (1) external software 450, which is composed of elements well known to the art and beyond the scope of the present invention, such as device drivers, the native operating systems (such

as D[illegible] and MAC Toolbox), machine-specific assembler functions (in the preferred embodiment, for example, CRC checking), and "C" runtime library functions; (2) native software 420; and (3) partitioned applications 410.

Again with reference to FIG. 7, native software 420 is compiled from the "C" language into a target machine-specific executable, and is composed of two components: the service software 430 and 431 and the operating environment 450. Operating environment 450 is comprised of the Logical Operating System 432, or LOS; and a multitasker 433. Service software 430 and 431 provides functions specific to providing interaction between the user and interactive network 10, while the operating environment 450 provides pseudo multitasking and access to local physical resources in support of service software 430 and 431. Both layers of native software 420 contain kernal, or device independent functions 430 and 432, and machine-specific or device dependent functions 431 and 433. All device-dependencies are in code resident at RS 400, and are limited to implementing only those functions that are not common across machine types, to enable interactive network 10 to provide a single data stream to all makes of personal computer. Source code for the native software accompanies this specification and provides a detailed teaching of the art concerning the claims directed toward the software comprising this invention.

Service software 430 and 431 is comprised of modules, which are device-independent software components that together obtain, interpret and store partitioned applications existing as a collection of objects. The functions performed by, and the relationship between, the service software 430 and 431 modules are described in FIG. 8 and discussed further below.

Through facilities provided by LOS 432 and multitasker 433, here called collectively operating environment 450, provides device-independent multitasking and access to local machine resources, such as multitasking, timers, buffer management, dynamic memory management, file storage and access, keyboard and mouse input, and printer output. The operating environment 450 manages communication and synchronization of service software 430 and 431, by supporting a request/response protocol and managing the interface between the native software 420 and external software 437.

Applications software layer 410 consists of programs and data written in an interpretive language, "TRINTEX Basic Object Language" or "TBOL," which was written specifically for use in RS 400 and interactive network 10 to facilitate videotex-specific commands and achieve machine-independent compiling. TBOL is the current preferred embodiment for this invention. TBOL is constructed as objects, which in interaction with one another comprise partitioned applications, and is disclosed at length elsewhere in this document.

RS native software provides a virtual machine interface for partitioned applications, such that all objects comprising partitioned applications "see" the same machine. RS native software provides support for the following functions: (1) keyboard and mouse input; (2) text and graphics display; (3) application interpretation; (4) application database management; (5) local application storage; (6) network and link level communications; (7) user activity data collection; and (8) advertisement management.

With reference to FIG.8, service software 430 and 431 is comprised of the following modules: (1) startup; (2) keyboard manger 434 ; (3) object interpreter 435 ; (4) TBOL interpreter 438; (5) object storage facility 439; (6) display manager 461; (7) data collection manager 466; (8) ad manager 432; (9) object/communications manager interface 433; (10) link communications manager 444; and (11) fatal error manager 469. Each of these modules has responsibility for managing a different aspect of RS 400.

Startup reads RS 400 customization options into RAM, including modem, device driver and telephone number options, from the file CONFIG.SM. Startup invokes all RS 400 component startup functions, including navigation to the first page, a logon screen display containing fields initialized to accept the user's id and password. Startup is not shown in FIG. 8.

The principal function of keyboard manger 434 is to translate personal computer dependent physical input into a consistent set of logical keys and to invoke processors associated with these keys. Depending on the LOS key, and the associated function attached to it, navigation, opening of windows, and initiation of filter or post-processor TBOL programs may occur as the result input events handled by the keyboard manger 434. In addition, keyboard manger 434 determines inter- and intra-field cursor movement, and coordinates the display of field text and cursors entered by the user with display manager 461, and sends information regarding such inputs to data collection manager 466.

Object interpreter 435 is responsible for building and recursively processing a table called the "Page Processing Table," or PPT. Object interpreter 435 also manages the opening and closing of windows at the current page. Object interpreter 435 is implemented as two subcomponents: the object processor 436 and object scanner 437.

Object processor 436 provides an interface to keyboard manger 434 for navigation to new pages, and for opening and closing windows in the current page. Object processor 436 makes a request to Object storage facility 439 for a PTO or WEO, as requested by keyboard manger 434, and for objects and their segments which comprise the PTO or WEO returned by Object storage facility 439 to object processor 436. Based on the particular segments comprising the object(s) making up the new PTO or WEO,

object processor 436 builds or adds to the PPT, which is an internal, linked-list, global data structure reflecting the structure of the page or PFO, each page partition or PEO, and Program Data Objects (PDOs) required and each window object that could be called. Objects are processed by parsing and interpreting each object and its segment(s) according to their particular structure as formalized in the DOA. While in the process object state, object processor 436 will request any objects specified by the PTO that are identified by external references in call segments (e.g. field level program call 518, page element selector call 524, page format call 526 program call 532, page element call 522 segments) of such objects, and will, through a request to TBOL interpreter 438, fire initializers and selectors contained in program data segments of all PTO constituent program objects, at the page, element, and field levels. Object processor 435 requests all objects required to build a page, except objects that could only be called as the result of some event external to the current partitioned application, such as a HELP window object. When in the course of building or adding to the PPT and opening/closing WEOs, object processor encounters a call to an object with object-id "ADSLOT," it fetches the next advertisement object 510 from ad manager 432, and sends to display manager 461 for display to the user presentation data segments 530 contained in the objects constituent of the PTO, WEO and advertisement object. Object processor also passes to data collection manager 466 all object-ids that were requested and object-ids that were viewed. Upon completion of page or window

processing, object processor 436 enters the wait for event state, and control is returned to keyboard manger 434.

The second component of object interpreter 435, object scanner 437, provides a file-like interface, shared with object storage facility 439, to objects currently in use at RS 400, to enable object processor 436 to maintain and update the PPT. Through facilities provided by object scanner 437, object processor recursively constructs a page or window in the requested or current partitioned application, respectively.

Object storage facility provides an interface through which object interpreter 435 and TBOL interpreter 438 either synchronously request (using the TBOL verb operator "GET") objects without which processing in either module cannot continue, or asynchronously request (using the TBOL verb operator "FETCH") objects in anticipation of later use. Object storage facility 439 returns the requested objects to the requesting module once retrieved from either local store 440 or interactive network 10. Through control structures shared with the object scanner 437, object storage facility determines whether the requested object resides locally, and if not, makes an attempt to obtain it from interactive network 10 through interaction with link communications manager 444 via object/communications manager interface 433.

When objects are requested from object storage facility 439, it must always provide the latest version of the object to guarantee currency of information to the user. Object storage facility 439 does this by requesting objects that are not locally available and by requesting version verification from network 10 for those objects which are available locally.

Version verification increases response time. Therefore, not all objects locally available are version checked each time they are requested. Typically, objects are checked only the first time they are requested during a user session.

The frequency with which the currency of objects is checked depends on factors such as the frequency of updating of the objects. For example, objects that are designated as ultrastable in a storage control parameter in the header of the object are never version checked unless a special version control object sent to the RS as part of logon indicates that all such objects must be version checked. Object storage facility 439 marks all object entries with such a stability category in all directories indicating that they must be version checked the next time they are requested.

Object storage facility 439 manages objects locally in local store 440, comprised of a cache (segmented between available RAM and a fixed size disk file), and stage (fixed size disk file). Cached objects are retained only during user sessions, while

stage objects are retained between sessions. The storage control field, located in the header portion of an object, indicates whether the object is stageable, cacheable or trashable.

Stageable objects must not be subject to frequent change or update. They are retained between user sessions on the system, provided storage space is available and the object is not discarded by a self-configuring stage algorithm, which is used to determine storage priority for the fixed size disk file. Over time, the self-configuring stage algorithm will have the effect of retaining within local disk storage those objects which the user has accessed most often. The objects retained locally are thus optimized to each individual user's usage of the applications in the system. Response time to such objects is optimized since they need not be retrieved from the interactive computer system. This is done for each individual user, whose storage is kept separate from other users of the same personal computer 405 by using separate diskettes or separate directories on a shared storage medium.

Cacheable objects can be retained during the current user session, but cannot be retained between sessions. These objects usually have a moderate update frequency. Object storage facility 439 retains objects in the cache according to a Least Recently Used (LRU) storage retention algorithm. Object storage facility 439 uses the LRU algorithm to ensure that objects that

are least frequently used forfeit their storage to objects that are more frequently used.

Trashable objects can be retained only while the user is in the context of the partitioned application in which the object was requested. Trashable objects usually have a very high update frequency and must not be retained to ensure that the user has access to the most current data.

TBOL interpreter 438 provides the means to interpretively execute program objects, which have been written using an interpretive language, TBOL (described elsewhere in this specification). TBOL interpreter 438 interprets operators and operands contained in program object 508, manages TBOL variables and data, maintains buffer and stack facilities, and provides a run-time library of TBOL verbs.

TBOL verbs provide support for data processing, program flow control, file management, object management, communications, text display, command bar 290 control, open/close window, page navigation and sound. TBOL interpreter also interacts with other native modules through commands contained in TBOL verbs. For example: the verb "navigate" will cause TBOL interpreter 438 to request object interpreter 435 to build a PPT based on the PTO-id contained in the operand of the NAVIGATE verb; "fetch" or "GET" will cause TBOL interpreter 438 to request an object from object storage facility 439; "SET_FUNCTION" will assign a filter to

events occurring at the keyboard manger 434; and "FORMAT," "SEND," and "RECEIVE" will cause TBOL interpreter 438 to send application-level requests to object/communications manager interface 433.

Data areas managed by TBOL interpreter 438 and available to TBOL programs are Global External Variables (GEVs), Partition External Variables (PEVs), and Runtime Data Arrays (RDAs).

GEVs contain global and system data, and are accessible to all program objects as they are executed. GEVs provide a means by which program objects may communicate with other program objects or with the RS native code, if declared in the program object. GEVs are character string variables that take the size of the variables they contain. GEVs may preferably contain a maximum of 32,000 variables and are typically used to store such information as program return code, system date and time, or user sex or-
age. TBOL interpreter 438 stores such information in GEVs when requested by the program which initiated a transaction to obtain these records from the RS or user's profile stored in the interactive system.

Partition external variables (PEVs) have a scope restricted to the page partition on which they are defined. PEVs are used to hold screen field data such that when PEOs and window objects are defined, the fields in the page partitions with which these

objects are to be associated are each assigned to a PEV. When applications are executed, TBOL interpreter 438 transfers data between screen fields and their associated PEV. When the contents of a PEV are modified by user action or by program direction, TBOL interpreter makes a request to display manager to update the screen field to reflect the change. PEVs are also used to hold partition specific application data, such as tables of information needed by a program to process an expected screen input.

Because the scope of PEVs is restricted to program objects associated with the page partition in which they are defined, data that is to be shared between page partitions or is to be available to a page-level processor must be placed in GEVs or RDAs.

RDAs are internal stack and save buffers used as general program work areas. RDAs are dynamically defined at program object "run-time" and are used for communication and transfer of data between programs when the data to be passed is not amenable to the other techniques available. Both GEVs and RDAs include, in the preferred embodiment, 8 integer registers and 8 decimal registers. Preferably, there also 9 parameter registers limited in scope to the current procedure of a program object.

All variables may be specified as operands of verbs used by the virtual machine. The integer and decimal registers may be

specified as operands for traditional data processing. The parameter registers are used for passing parameters to "called" procedures. The contents of these registers are saved on an internal program stack when a procedure is called, and are restored when control returns to the "calling" procedure from the "called" procedure.

TBOL interpreter, keyboard manger 434, object interpreter 435, and object storage facility 439, together with device control provided by operating environment 450, have principal responsibility for the management and execution of partitioned applications at the RS 400. The remaining native code modules function in support and ancillary roles to provide RS 400 with the ability display partitioned applications to the user (display manager 461), display advertisements (ad manager 432), to collect usage data for distribution to interactive network 10 for purposes of targeting such advertisements (data collection manager 466), and prepare for sending, and send, objects and messages to interactive network 10 (object/communications manager interface 433 and link communications manager 444) Finally, the fatal error manager exists for one purpose: to inform the user of RS 400 and transmit to interactive network 10 the inability of RS 400 to recover from a system error.

Display manager 461 interfaces with a decoder using the North American Presentation Level Protocol Syntax (NAPLPS), a standard for encoding graphics data, or text code, such as ASCII,

which are displayed on the monitor of the user's personal computer 405 as pictorial codes. Codes for other presentation media, such as audio, can be specified by using the appropriate type code in the presentation data segments. Display manager 461 supports the following functions: send NAPLPS strings to the decoder; echo text from a PEV; move the cursor within and between fields; destructive or non-destructive input field character deletion; "ghost" and "un-ghost" fields (a ghosted field is considered unavailable, un-ghosted available); turn off or on the current field cursor; open, close, save and restore bit-maps for a graphics window; update all current screen fields by displaying the contents of their PEVs, reset the NAPLPS decoder to a known state; and erase an area of the screen by generating and sending NAPLPS to draw a rectangle over that area. Display manager 461 also provides a function to generate a beep through an interface with a machine-dependent sound driver.

Ad manager 432 is invoked by object interpreter 435 to return the object-id of the next of the next available advertisement to be displayed. Ad manager 432 maintains a queue of advertisement object-id's targeted to the specific user currently accessing interactive network 10. Advertisement objects 510 are pre-fetched from interactive system 10 from a personalized queue of advertisements that is constructed using data previously collected from user generated events and/or reports of objects used in the building of pages or windows, compiled by data collection manager 466 and transmitted to interactive system 10.

Advertisement objects 510 are PEOs that, through user invocation of a "LOOK" command, cause navigation to partitioned applications that may themselves support, for example, ordering and purchasing of merchandise.

An advertisement list, or "ad queue," is requested in a transaction message to delivery system 20 by ad manager 432 immediately after the initial logon response. The logon application at RS 400 places the advertisement list in a specific RS global storage area called a SYS_GEV (system global external variable), which is accessible to all applications as well as to the native RS code). The Logon application also passes the first two ad object-id's to object storage facility 439 to be requested. At logon, no advertisement objects will be available RS local storage facilities 440, so they must be requested from interactive system 10.

In a preferred embodiment, the following parametric values are established for ad manager 432: advertisement queue capacity, replenishment threshold for advertisement object-id's and replenishment threshold for number of outstanding pre-fetched advertisement objects. These parameters are set up in GEVs of the RS virtual machine by the logon application program object from the logon response from high function system 110. The parameters are then also accessible to the ad manager 432. Preferred values are an advertisement queue capacity of 15,

replenishment value of 10 empty queue positions and a pre-fetched advertisement threshold of 3.

Ad manager 432 pre-fetches advertisement object 510 by passing advertisement object-id's from the advertisement queue to object storage facility 439 which then retrieves the object from the interactive system if the object is not available locally. Advertisements are pre-fetched, so they are available in RS local store 440 when requested by object-id by object interpreter 435 while it is building a page. The Ad manager 432 pre-fetches additional advertisement objects whenever the number of pre-fetched advertisements, not used by object interpreter 435 falls below the pre-fetch advertisement threshold.

Whenever the advertisement queue has more empty positions than replenishment threshold, a transaction is made to the advertisement queue application in high function system 110 via object/communications manager interface 433 for a number of advertisement object-id's equal to the threshold. A response message includes a list of advertisement object-id's, which ad manager 432 enqueues.

Object interpreter 435 requests the object-id of the next advertisement from ad manager 432 when object interpreter 435 is building a page and encounters an object call for a partition and the specified object-id equals the code word, "ADSLOT." If this is the first request for an advertisement object-id that ad

manager 432 has received during this user's session, ad manager 432 moves the advertisement list from the GEV into its own storage area, which it uses as an advertisement queue and sets up its queue management pointers, knowing that the first two advertisement objects have been pre-fetched.

Ad manager 432 then queries object storage facility 439, irrespective of whether it was the first request of the session. The query asks if the specified advertisement object id pre-fetch has been completed, i.e., is the object available locally at the RS. If the object is available locally, the object-id is passed to object interpreter 435, which requests it from object storage facility 439. If the advertisement object is not available in local store 440, ad manager 432 attempts to recover by asking about the next ad that was pre-fetched. This is accomplished by swapping the top and second entry in the advertisement queue and making a query to object storage facility 439 about the new top advertisement object-id. If that object is not yet available, the top position is swapped with the third position and a query is made about the new top position.

Besides its ability to provide advertisements that have been targeted to each individual user, two very important response time problems have been solved by ad manager 432 of the present invention. The first is to eliminate from the new-page response time the time it takes to retrieve an advertisement object from

the host system. This is accomplished by using the aforementioned pre-fetching mechanism.

The second problem is caused by pre-fetching, which results in asynchronous concurrent activities involving the retrieval of objects from interactive system 10. If an advertisement is pre-fetched at the same time as other objects required for a page requested, the transmission of the advertisement object packets could delay the transmission of the other objects required to complete the current page by the amount of time required to transmit the advertisement object(s). This problem is solved by the structuring the requests from object interpreter 435 to the ad manager 432 in the following way:

1. Return next object-id of pre-fetched advertisement object & pre-fetch another;
2. Return next advertisement object-id only;
3. Prefetch next advertisement object only.

By separating the function request (1) into its two components, (2) and (3), object interpreter 435 is now able to determine when to request advertisement object-id's and from its knowledge of the page build process, is able to best determine when another advertisement object can be pre-fetched, thus causing the least impact on the page response time. For example, by examining the PPT, object interpreter 435 may determine whether any object requests are outstanding. If there are

outstanding requests, advertisement request type would be used. When all requested objects are retrieved, object interpreter 435 then issues an Advertisement request type 3. Alternatively, if there are no outstanding requests, object interpreter 435 issues an advertisement request type 1. This typically corresponds to the user's "think time" while examining the information presented and when RS 400 is in the Wait for Event state (D).

Data collection manager 466 is invoked by object interpreter 435 and keyboard manger 434 to keep records about what objects a user has obtained (and, if a presentation data segment 530 is present, seen) and what actions users have taken (e.g. "NEXT," "BACK," "LOOK," etc.)

The data collection events that are to be reported during the user's session are sensitized during the logon process. The logon response message carries a data collection indicator with bit flags set to "on" for the events to be reported. These bit flags are enabled (on) or disabled (off) for each user based on information contained in the user's profile stored and sent from high function host 110. A user's data collection indicator is valid for the duration of his session. The type of events to-be reported can be changed at will in the host data collection application. However, such changes will affect only users who logon after the change.

Data collection manager 466 gathers information concerning a user's individual system usage characteristics. The types of informational services accessed, transactions processed, time information between various events, and the like are collected by data collection manager 466, which compiles the information into message packets (not shown). The message packets are sent to interactive system 10 via object/communication manager interface 433 and link communications manager 444. Message packets are then stored by high function host 110 and sent to an off-line processing facility for processing. The characteristics of users are ultimately used as a means to select or target various display objects, such as advertisement objects, to be sent to particular users based on consumer marketing strategies, or the like, and for system optimization.

Object/communications manager interface 433 is responsible for sending and receiving DIA (Data Interchange Architecture -- described elsewhere in this document) formatted messages to or from interactive network 10. Object/communications manager also handles the receipt of objects, builds a DIA header for messages being sent and removes the header from received DIA messages or objects, correlates requests and responses, and guarantees proper block sequencing. Object/communications manager interface 433 interacts with other native code modules as follows: object/communications manager (1) receives all RS 400 object requests from object storage facility 439, and forwards objects received from network 10 via link communications manager 444

directly to the requesting modules; (2) receives ad list requests from ad manager 432, which thereafter periodically calls object/communications manager to receive ad list responses; (3) receives data collection messages and send requests from data collection manager; (4) receives application-level requests from TBOL interpreter 438, which also periodically calls object/communications manager interface 433 to receive responses (if required); and (5) receives and sends DIA formatted objects and messages from and to link communications manager 444.

Object/communications manager interface 433 sends and receives DIA formatted messages on behalf of TBOL interpreter 438 and sends object requests and receives objects on behalf of object storage facility 439. Communication packets received containing parts of requested objects are passed to object storage facility 439 which assembles the packets into the object before storing it. If the object was requested by object interpreter 435, all packets received by object storage facility 439 are also passed to object interpreter 435 avoiding the delay required to receive an entire object before processing the object. Objects which are pre-fetched are stored by object storage facility 439.

Messages sent to interactive system 10 are directed via DIA to applications in interactive system 10. Messages may include transaction requests for records or additional processing of records or may include records from a partitioned application

program object or data collection manager 466. Messages to be received from interactive system 10 usually comprise records requested in a previous message sent to interactive system 10. Requests received from object storage facility 439 include requests for objects from storage in interactive system 10. Responses to object requests contain either the requested object or an error code indicating an error condition.

Object/communications manager 433 is normally the exclusive native code module to interface with link communications manager 444 (except in the rare instance of a fatal error). Link communications manager 444 controls the connecting and disconnecting of the telephone line, telephone dialing, and communications link data protocol. Link communications manager 444 accesses interactive system 10 by means of a communications medium (not shown) link communications manager 444, which is responsible for a dial-up link on the public switched telephone network (PSTN). Alternatively, other communications means, such as cable television or broadcast media, may be used. Link communications manager 444 interfaces with TBOL interpreter for connect and disconnect, and with interactive network 10 for send and receive.

Link communications manager 444 is subdivided into modem control and protocol handler units. Modem control (a software function well known to the art) hands the modem specific handshaking that occurs during connect and disconnect. Protocol handler is responsible for transmission and receipt of data

packets using the TCS (TRINTEX Communications Subsystem) protocol (which is a variety of OSI link level protocol, also well known to the art).

Fatal error manager 469 is invoked by all reception system components upon the occurrence of any condition which precludes recovery. Fatal error manager 469 displays a screen to the user with a textual message and an error code through display manager 461. Fatal error manager 469 sends an error report message through the link communications manager 444 to a subsystem of interactive network 10.

NOTE REGARDING SOURCE CODE: All of the source code for RS 400 is provided as part of this specification. Nomenclature for the various service software 430 and 431 modules may differ, but the functions described herein are implemented in the source code. Some functions described herein are implemented across modules in source code. The following is a concordance of the terms used in this section of the disclosure and the terms used in the source code:

Specification		Source Code
Keyboard Manager	=	Input Manager/Event Processor
Object Interpreter	=	Object Processor

TBOL Interpreter	=	API or Logic Interpreter (the above 3 modules are referred to as the Service Manager)
Object Storage Facility	=	Object Manager
Object/Communications Manager	=	Message Manager and Communications Manager
Ad Manager	=	Ad Manager
Display Manager	=	Display Manager
Data Collection Manager	=	Data Collection Manager
Link Communications Manager	=	Communications Manager

SOURCE CODE ORGANIZATION The source code for the reception system 400 software is printed on the pages that follow. All files within each of the MS-DOS directories listed below are printed on sequentially numbered pages bearing the pathname of the directory to which they belong.

rs

api

c

inc

applib

asm

c

inc

cm

asm

c

inc

esp

asm

c

inc

los

asm

c

inc

oversutl

c

inc

rsk

asm

c

inc

sm

asm

```
             c
             inc
storeutl
             c
             inc
tmk
             asm
             c
             inc
ver_esp
             c
             inc
ver_over
             c
             inc
ver_sm
             c
             inc
ver_stor
             c
             inc
```

SAMPLE APPLICATION

The page illustrated in FIG. 3(b) corresponds to a partitioned application that permit's a personal computer user to purchase apples. It shows how the monitor screen of personal computer 405 might appear to the user. The displayed page includes a number of page partitions and corresponding page elements.

The PTO 500 representing this page 280 is illustrated in FIG. 9. PTO 500 defines the composition of the page, including header 250, body 260, display fields 270, advertisement 280, and command bar 290. PEOs 504 are associated with page partitions numbered #1, #2 and #3. They present information in the header 250, identifying the page topic as ABC APPLES; in the body 260, identifying the cost of apples; and prompt the user to input into fields within body 260 the desired number of apples to be ordered. In advertisement 280, presentation data and a field representing a post-processor that will cause the user to navigate to a targetable advertisement, is presented.

In FIG. 9, the structure of the PTO 500 can be traced. PTO 500 contains a page format call segment 526, which calls PFO 502. PFO 502 describes the location and size of partitions on the page and numbers assigned to each partition. The partition number is used in page element call segments 522 so that an association is established between a called PEO 504 and the page partitionwhere

it to be displayed. Programs attached to this PEO can be executed only when the cursor is in the page partition designated within the PEO.

PTO 500 contains two page element call segments 522, which reference the PEOs 504 for partitions #1 and #2. Each PEO 504 defines the contents of the partition. The header in partition #1 has only a presentation data segment(s) 530 in its PEO 504. No input, action, or display fields are associated with that partition.

The PEO 504 for partition #2 contains a presentation data segment 530 and field definition segments 516 for the three fields that are defined in that partition. Two of the fields will be used for display only. One field will be used for input of user supplied data.

In the example application, the PEO 504 for body partition #2 specifies that two program objects 508 are part of the body partition. The first program, shown in Display field 270 is called an initializer and is invoked unconditionally by TBOL interpreter 438 concurrently with the display of presentation data for the partition. In this application, the function of the initializer is represented by the following pseudo-code:

1. Move default values to input and display fields; 2. "SEND" a transaction to the apple application that

is resident on interactive system 10;

3. "RECEIVE" the result from interactive system 10 - i.e. the current price of an apple;

4. Move the price of an apple to PEV 270 number 1 so that it will be displayed;

5. Position the cursor on the input field; and

6. Terminate execution of this logic.

The second program object 508 is a field post-processor. It will be invoked conditionally, depending upon the user keystroke input. In this example, it will be invoked if the user changes the input field contents by entering a number. The pseudo-code for this post-processor is as follows:

1. Use the value in PEV 270 number 2 (the value associated with the data entered by the user into the second input data field 270) to be the number of apples ordered.

2. Multiply the number of apples ordered times the cost per apple previously obtained by the initializer;

3. Construct a string that contains the message "THE COST OF THE APPLES YOU ORDERED IS $45.34;";

4. Move the string into PEV 270 number 3 so that the result will be displayed for the user; and

Terminate execution of this logic.

The process by which the "APPLES" application is displayed, initialized, and run is now discussed.

The "APPLES" application is initiated when the user navigates from the previous partitioned application, with the navigation target being the object-id of the "APPLES" PTO 500 (that is, object-id ABC1). This event causes keyboard manager 434 to pass the PTO object-id, ABC1 (which may, for example, have been called by the keyword navigation segment 520 within a PEO 504 of the previous partitioned application), to object interpreter 435. With reference to the RS application protocol depicted in FIG. 6, when the partitioned application is initiated, RS 400 enters the Process Object state (B) using transition (1). Object interpreter 435 then sends a synchronous request for the PTO 500 specified in the navigation event to object storage facility 439. Object storage facility 439 attempts to acquire the requested object from local store 440 or from delivery system 20 by means of object/communication manager 443, and returns an error code if the object cannot be acquired.

Once the PTO 500 is acquired by object/communications manager 443, object interpreter 435 begins to build PPT by parsing PTO 500 into its constituent segment calls to pages and page elements, as shown in FIG. 4d and interpreting such segments. PFO and PEO call segments 526 and 522 require the

acquisition of the corresponding objects with object-id's <ABCF>, <ABCX> and <ABCY>. Parsing and interpretation of object ABCY requires the further acquisition of program objects <ABCI> and <ABCJ>.

During the interpretation of the PEOs 504 for partitions 1 and 2, other RS 400 events are triggered. This corresponds to transition (2) to Interpret Pre-processors state (C) in FIG. 6. Presentation data 530 is sent to display manager 461 for display using NAPLPS decoder 476, and, as the PEO <ABCY> for partition #2 is parsed and interpreted by object interpreter 435, parameters in program call segment 532 identify the program object <ABCI> as an initializer. Object interpreter 435 obtains the program object from object storage facility 439, and makes a request to TBOL interpreter 438 to execute the initializer program object 508 <ABCI>. The initializer performs the operations specified above using facilities of the RS virtual machine. TBOL interpreter 438, using operating environment 450, executes initializer program object 506 <ABCI>, and may, if a further program object 508 is required in the execution of the initializer, make a synchronous application level object request to object storage facility 439. When the initializer terminates, control is returned to object interpreter 435, shown as the return path in transition (2) in FIG. 6.

Having returned to the Process Object state (B), object processor 435 continues processing the objects associated with

PTO <ABC1>. Object interpreter continues to construct the PPT, providing RS 400 with an environment for subsequent processing of the PTO <ABC1> by pre-processors and post-processors at the page, partition, and field levels. When the PPT has been constructed and the initializer executed, control is returned to keyboard manager 434, and the RS enters the Wait For Event (E) State, via transition (4), as shown in FIG. 6.

In the Wait for Event state, the partitioned application waits for the user to create an event. In any partitioned application, the user has many options. For example, the user may move the cursor to the "JUMP" field on the command bar, which is outside the current application, and thus cause subsequent navigation to another application. For purposes of this example, it is assumed that the user enters the number of apples he wishes to order by entering a digit in display field 270 number 2.

Keyboard manager translates the input from the user's keyboard or mouse, which varies according to the type of personal computer used, to a logical representation independent of any type of personal computer. Keyboard manager 434 saves the data entered by the user in a buffer associated with the current field defined by the location of the cursor. The buffer is indexed by its PEV number, which is the same as the field number assigned to it during the formation of the page element. Keyboard manager 434 determines for each keystroke whether the keystroke co-rresponds to an input event or to an action or completion event.

Input events are logical keystrokes and are sent by event processor 454 to display manager 437, which displays the data at the input field location. Display manager 437 also has access to the field buffer as indexed by its PEV number.

The input data are available to TBOL interpreter 438 for subsequent processing. When the cursor is in a partition, only the PEVs for that partition are accessible to the RS virtual machine. After the input from the user is complete (as indicated by a user action such as pressing the RETURN key or entry of data into a field with an action attribute), RS 400 enters the Process Event state (E) via transition (4).

For purposes of this example, let us assume that that the user enters the digit "5" in input field 3. A transition is made to the Process Event state (E). Event processor 454 and display manager 437 perform a number of actions, such as the display of the keystroke on the screen, the collection of the keystroke for input, and optionally, the validation of the keystroke, i.e. numeric input only in numeric fields. When the keystroke is processed, a return is made to the Wait for Event state (D). Edit attributes are specified in the field definition segment.

Suppose the user inputs a "6" next. A transition occurs to the PE state and after the "6" is processed, the Wait for Event (D) state is re-entered. If the user hits the "completion" key (e.g.,ENTER) the Process Event (E) state will be entered. The

action attributes associated with field 3 identify this as a system event to trigger post-processor program object <ABCJ>. When the interpretive execution of program object <ABCJ> is complete, the Wait for Event state (D) will again be entered. The user is then free to enter another value in the input field, or select a command bar function and exit the apples application.

B2 Amd'd

At Page 57, after Line 2, add the following pages of Source Code, which for convenience have been collected in 36 sections, each identified at the upper right hand corner of the first page of each section and consecutively numbered beginning at "1" therein.

(Amendments to the claims begin at Page 235 of the Preliminary Amendment)